MARCH/APRIL 2016

Java™ magazine

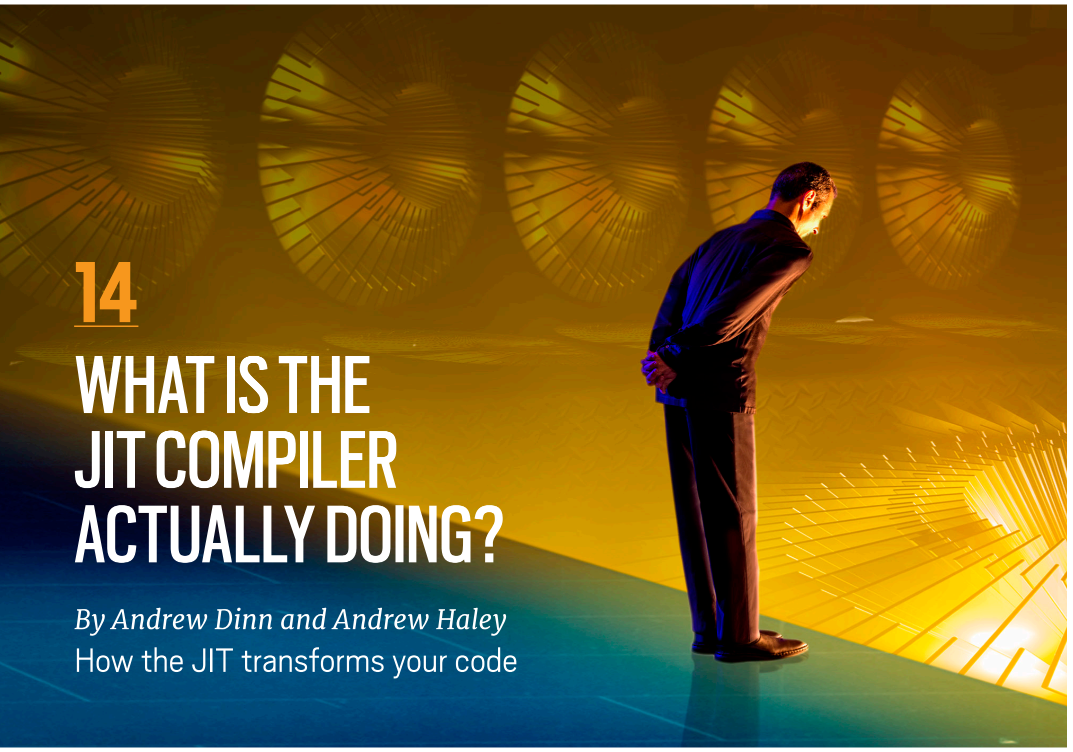
By and for the Java community

Inside Java and the JVM

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /



EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Copy Editors
Karen Perkins, Jim Donahue

Section Development
Michelle Kovac

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Production Manager
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Tom Cometa tom.cometa@oracle.com

Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

Greater Type Inference and Brevity May Be Coming to Java

A proposal for new reserved words will cut boilerplate code.

JVM languages can be divided into two broad categories: those that aim to improve on Java's design (Groovy, Kotlin, Scala, Golo, Gosu, and so on) and those that are ports of other languages to the Java platform (JRuby, Jython, Fortress, and others). In the former group, three enhancements are almost universal: concision, closures, and simple ways of specifying immutability. It would be tempting (although not completely accurate) to restate these three differentiators as a quest for brevity, but the more precise way to say it would be that they are all part of a quest for *simplicity*.

In this regard, the last two releases of Java have seen important changes that move the language toward this goal of reduced clutter. In particular, the advent of lambdas and the introduction of streams in Java 8 shrank the amount of code previously necessary to communicate straightforward operations.

However, those changes did not fully attack the much-assailed wordiness of the language. This loquaciousness is most visible and annoying in type declarations:

```
HaydnSymphony surprise =
    new HaydnSymphony();
```

It's clear in this example that the compiler knows the type of the declared item, so it's rather pointless to type it twice. And in enterprise applications (where the naming problem is particularly acute), variables can have lengthy names, which make not only writing but reading code tiring and error-prone. To address the question of brevity,

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

the problem of redundant type declarations needed to be resolved.

A new JDK Enhancement Proposal, JEP 286, proposes to do just this for local variables. While several approaches are considered, the one actually proposed puts forth a new reserved word, `var`, which stands in for the redundant boilerplate. The previous code would now look somewhat shorter:

```
var surprise =
    new HaydnSymphony();
```

It's important to note that `var` is just a reserved word here, not some declaration of a new dynamic type. It simply states that the type, which remains static, can be inferred by the compiler on the right-hand side of the declaration.

By wisely relegating this feature to local variables, the JEP authors made sure the actual data type is never far from where the variable is used, so downstream maintenance programmers will find it easy to know exactly what the data type is. This aspect is important so that `var` does not accidentally make it difficult to understand or debug code.

Had the JEP proposed only this, it would be a welcome step forward. But it goes even further and entertains the possibility of a second keyword that would allow similar declaration syntax for immutable objects. If Java were to use the scheme for variables and values found in Scala, the second keyword would be `val` (for *value*). It would look like this:

```
val normalTemp = 98.6;
```

Because the compiler can tell that the initial value is a floating-point constant, it can easily infer the correct data type.

The proposed type inference of `var` extends the type inference previously delivered in Java 5, 7, and 8; whereas the use of `val` is principally a replacement for `final`. In fact, Java already has a second keyword, `const`, which is currently reserved but unused, that could be employed for the same purpose. In a world of abstractions, the idea of a third reserved term for the same concept might seem preposterous, but I like the pragmatism of a short, elegant term as an immutable alternative to `var`. This similarity makes the language feel cohesive (compared with the more inchoate feel of C++).

The use of `var` and its possible pairing with `val` strongly appeal to me, but the JEP document explores other possible combinations. One of the suggested objections to `val` and `var` is that the similarity in spelling can lead to confusion, although I am not persuaded that this is a true problem. Of the complaints that Scala developers have about the language's syntax, confusion between these terms is not among them.

Adding new reserved words to an established language is serious business. Especially with short words, it's likely that the additions will cause disruptions in existing code. So additions should be made only when they address a compelling need. I believe that the brevity and simplicity these terms will bring to Java warrant this step, and I would love to see adoption of this proposal.

Let me know if you agree.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

CREATE THE FUTURE
oracle.com/java

Java™
ORACLE®

Reach More than 640,000 Oracle Customers with Oracle Publishing Group

Connect with the Audience that Matters Most to Your Business

Oracle Magazine
The Largest IT Publication in the World
Circulation: 325,000
Audience: IT Managers, DBAs, Programmers, and Developers

Profit
Business Insight for Enterprise-Class Business Leaders to Help Them Build a Better Business Using Oracle Technology
Circulation: 90,000
Audience: Top Executives and Line of Business Managers

Java Magazine
The Essential Source on Java Technology, the Java Programming Language, and Java-Based Applications
Circulation: 225,00 and Growing Steady
Audience: Corporate and Independent Java Developers, Programmers, and Architects

JANUARY/FEBRUARY 2016

How Groovy Found Its Groove

Thank you for mentioning Groovy's surge in popularity in your editorial, "The Rise and Fall of Languages in 2015" (January/February 2016, page 3). You attributed the recent success to performance. And indeed Groovy has moved from a dynamic Java companion to an efficient general-purpose language, with static type checking for refined type safety, and static compilation for Java speed and efficiency. So Groovy can be applied to any kind of programming activity.

This has led it to be more widely adopted in new use cases. For example, Groovy now has Android support, so you can use it for developing mobile apps.

Gradle, the popular build automation solution, uses Groovy as its build language, which makes Gradle more advanced and more flexible than other build products. Google adopted Gradle for building Android applications. So lots of Android developers—even those just using Java for their projects—are now also using Groovy, at least through Gradle, and so are being exposed and becoming familiar with the language.

Finally, I should note that Groovy is now part of the Apache Software Foundation and no longer directed by a commercial company. As a result of this change, other projects in the Apache Foundation are already using or integrating Groovy into their offerings.

—Guillaume Laforge
Project lead of the Groovy programming language

Elixir for Quirky Syntax

Thanks for your discussion of the various languages in your editorial, "The Rise and Fall of Languages in 2015." Even though you don't mention it, I'd be curious to get your thoughts on Elixir.

—Alan Andrade

Editor Andrew Binstock: Elixir is a new language that runs on top of Erlang on the Erlang virtual machine (called BEAM). It has been championed by Dave Thomas, whose early promotion of Ruby via his writings made that language popular. Erlang, a functional language with a quirky syntax, is designed for writing distributed applications that are fault tolerant. Elixir makes that Erlang syntax more approachable, while providing the same fault tolerance, due to running on BEAM and relying on the Erlang ecosystem. While I expect that Elixir will stay confined to the traditional niche that Erlang serves, I would not be surprised to see it overtake Erlang in popularity.

Testing Spring

Thank you for your informative article on Spring Boot ("First Steps with Spring Boot," January/February 2016, page 15). I had difficulty getting the test code on pages 20 and 21 to work. Ultimately, though, I was able to find the solution, which was to add the import statements in Listing 1 to the test code.

—Dave Brooks

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
```

Listing 1.

Jython Correction

I believe there are two minor errors in the Jython code examples in "Jython 2.7: Integrating Python and Java" (November/December 2015, page 42). In Listing 9, the fifth line should refer to `range2` rather than `range1`:

```
sum(get_nums(spreadsheet,
    range2)))
```

And in Listing 10, the range of spreadsheet cells being tested should not include `H5`:

```
assert_crosstab(main_sheet,
    "A5:G5", "H1:H4")
```

—Mihoko Suzuki
[Mihoko Suzuki heads up translation of *Java Magazine* into Japanese. —*Ed.*]

Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, indicate this in your message. Write to us at javamag_us@oracle.com. For other ways to reach us, including information on contacting customer service for your subscription, see the last page of this issue.

IJ

IntelliJ IDEA

CAPABLE AND
ERGONOMIC JAVA* IDE

Get it now

*Actually, much more than just Java

JET
BRAINS

Devoxx France *APRIL 20–22*
PARIS, FRANCE
Inaugurated by the Paris Java User Group in 2012 and open to all developers, Devoxx France will take place at the Palais des Congrès this year, with an estimated 2,500 participants and 220 presentations. The first day will be devoted to hands-on labs and three intensive tools-in-action presentations. The remaining days will follow the familiar regional Devoxx format featuring multiple miniconferences and small-group workshops. Keynote topics this year focus on computers in society.

Riga Dev Day
MARCH 2–4
RIGA, LATVIA
This event is a joint project by Google Developer Group Riga, Java User Group Latvia, and Oracle User Group Latvia. By and for software developers, Riga Dev Day focuses on 25 of the most relevant topics and technologies for that audience. Tracks include JVM and web development, databases, DevOps, and case studies.

PHOTOGRAPH BY TOMMIE HANSEN/FLICKR

EclipseCon
MARCH 7–10
RESTON, VIRGINIA
EclipseCon is all about community. Contributors, adopters, extenders, service providers, consumers, and business and research organizations gather to share their expertise and learn from each other. Topics this year include an introduction to the Eclipse Che next-generation Java IDE, hawkBit and software updates for the Internet of Things (IoT), faster index for Java, and Java 9 support in Eclipse.

O'Reilly Fluent Conference
MARCH 7–10
SAN FRANCISCO, CALIFORNIA
Fluent offers practical training in JavaScript, HTML5, CSS, and the latest web development technologies and frameworks. Topics include WebGL, CSS3, mobile APIs, Node.js, AngularJS, ECMAScript 6, and more. The conference is designed to appeal to application, web, mobile, and interactive developers, as well as engineers, architects, and UI/UX designers.

QCon London
MARCH 7–9, CONFERENCE
MARCH 10–11, WORKSHOPS
LONDON, ENGLAND
QCon is designed for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams. Topics include what to expect in Java 9 and Spring 5, containers in production, microservices for mega-architectures, full-stack JavaScript, and data science and machine learning methods. Two days of workshops follow the conference.

jDays
MARCH 8–9
GOTHENBURG, SWEDEN
jDays is a Java developer conference covering Java/Java EE, architecture, security, DevOps, cloud and microservices, testing, JavaScript, IoT trends, methodologies, and tools.

JavaLand 2016
MARCH 8–10
BRÜHL, GERMANY
This annual conference is a gathering of Java enthusiasts, developers, architects, strate-

PHOTOGRAPH BY GÜNTER HENTSCHEL/FLICKR

gists, and project administrators. Session topics for 2016 include containers and microservices, core Java and JVM languages, enterprise Java and the cloud, front end and mobile, IDEs and tools, and the IoT. After lectures on the first day of the conference, attendees get exclusive use of Phantasialand and its rides and attractions.

JAX 2016

APRIL 19–21, CONFERENCE
APRIL 18 AND 22, WORKSHOPS
MAINZ, GERMANY
More than 200 internationally renowned speakers give practical and performance-oriented lectures on topics such as Java, Scala, Android, web technologies, agile development models, and DevOps. Workshops are offered the day preceding and the day following the conference.

Great Indian Developer Summit (GIDS)

APRIL 26–30
BANGALORE AND PUNE, INDIA
The conference begins in Bangalore on April 26 through 29 and wraps up with an intensive one-day session in Pune on April 30. Tracks this year will focus on .NET and cloud development, web and mobile technologies, Java and dynamic languages, and data and analytics.

GeeCON 2016

MAY 11–13
KRAKOW, POLAND
GeeCON is a conference focused on Java and JVM-based technologies, with special attention to dynamic languages such as Groovy and Ruby. The event covers topics such as software development methodologies, enterprise architectures, software craftsmanship, design patterns, distributed computing, and more.

JEEConf 2016

MAY 20–21
KIEV, UKRAINE
JEEConf is the largest Java conference in Eastern Europe. The annual conference focuses on Java technologies for application development. This year offers five tracks and 45 speakers on modern approaches in the development of distributed, highly loaded, scalable enterprise systems with Java, among other topics.

jPrime

MAY 26–27
SOFIA, BULGARIA
jPrime is a relatively new conference with talks on Java, various languages on the JVM, mobile, web, and best practices. This second edition will be held in the Sofia Event Center, run by the Bulgarian Java User Group, and backed by the biggest companies in the city.

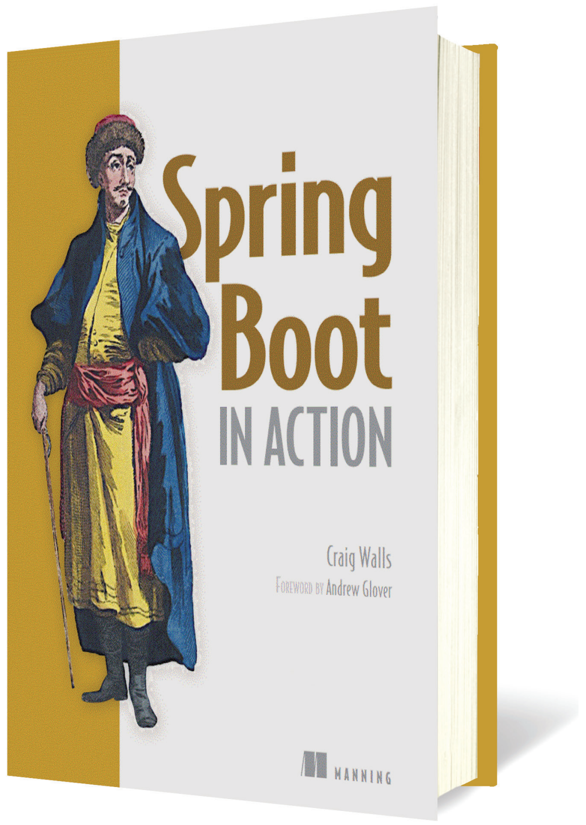
IndicThreads

JUNE 3–4
PUNE, INDIA
IndicThreads enters its 10th year featuring sessions on the latest in software development techniques and technologies from the IoT to big data, Java, web, and more.

Devoxx UK

JUNE 8–10
LONDON, ENGLAND
Devoxx UK focuses on Java, web, mobile, and JVM languages. The conference includes more than 100 sessions, with 50-minute conference sessions, three-hour hands-on labs, and many quickie presentations.

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event at least four months in advance at javamag_us@oracle.com. We'll include as many as space permits.



SPRING BOOT IN ACTION

By Craig Walls
Manning Publications

Spring Boot is the popular framework for Java enterprise development. It is based on the larger Spring Framework, and it enables developers to put together apps with less ceremony and housekeeping than found in most frameworks. The January/February issue of *Java Magazine*, which covered web development, had a feature article that explained the basics of setting up and running a Spring Boot application ("First Steps with Spring Boot").

This book, which at 180 pages plus appendixes is unusually short for a book in the Manning "in Action" series, presents an introduction similar to the article and then keeps going with more elaborate setups, more advanced configurations, and practical deployment information. The deployment coverage includes a discussion of pushing new apps to the cloud (specifically, Pivotal's Cloud Foundry and, in passing, Heroku). The explanation of the Spring ecosystem and how to use it in developing apps is approachable and intelligently presented.

As a starter manual, this book is satisfactory, but it contains several frustrating aspects of which these two in particular stand out. The first is that the downloadable source code is in places significantly different from what's printed in the book. What, then, is the reader to do or conclude? The later code (that online) is presumably more correct, but the text no longer corresponds directly to it. A second frustration is that the topic of microservices is not mentioned, even though this is clearly a major direction for Spring applications. Finally, I have difficulty with the author's penchant for Grails, a JVM application platform written in Groovy. While Grails 3.0 apps are based on Spring Boot, it's difficult to imagine that a Grails user would be turning to this book for information on Grails; meanwhile, a Spring Boot user is unlikely to want information about programming for a different application framework. Given the brevity of this book, it seems that other topics, such as microservices, would have been much more helpful.

Where the content is relevant and the code matches the printed volume, *Spring Boot in Action* is useful, well written, and easy to follow. If it were not for these limitations, it could be recommended.
—Andrew Binstock

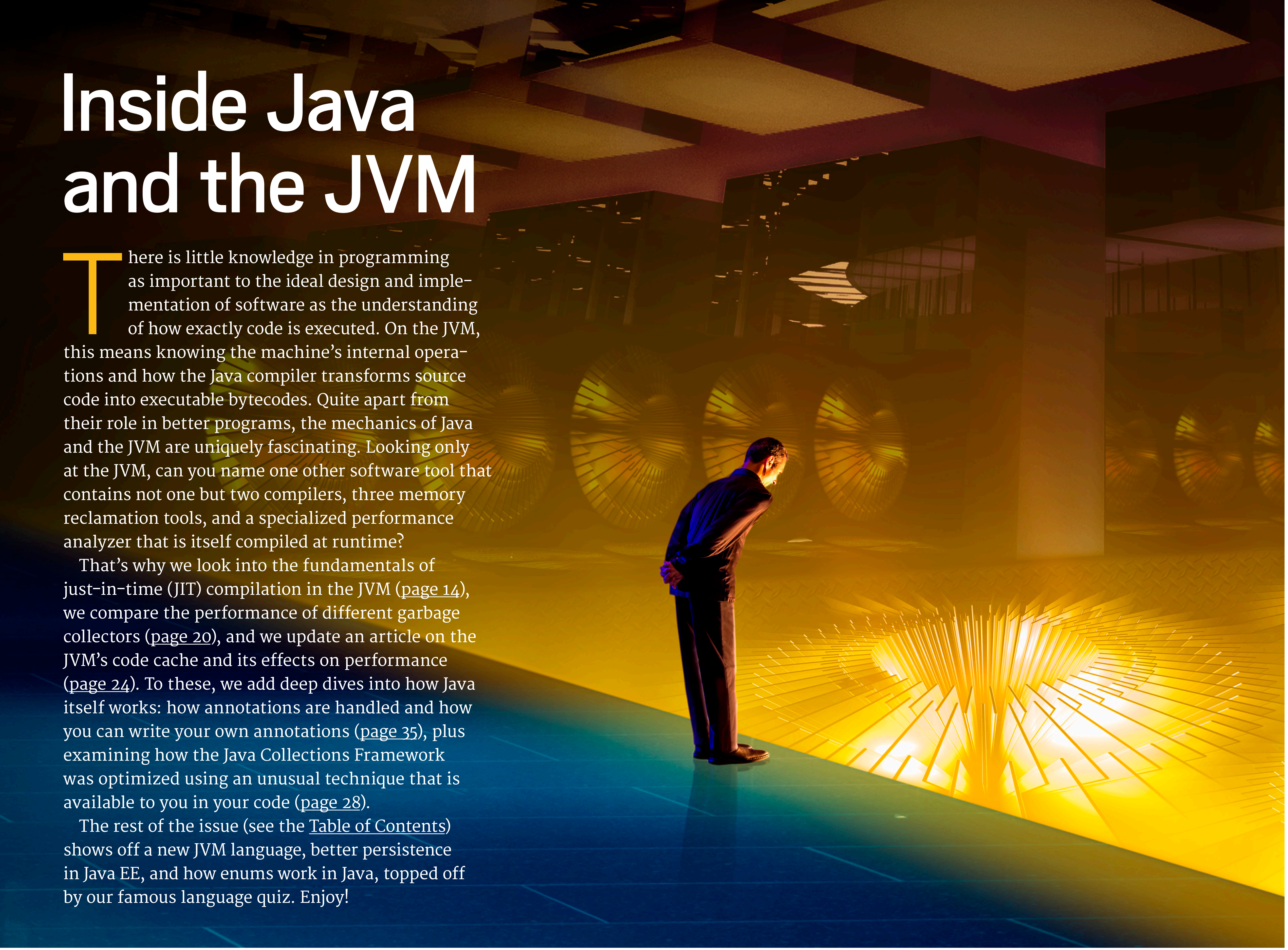
ZEROTURNAROUND

XRebel

THE LIGHTWEIGHT JAVA PROFILER

TRY IT FREE NOW!

Get a free t-shirt!



Inside Java and the JVM

There is little knowledge in programming as important to the ideal design and implementation of software as the understanding of how exactly code is executed. On the JVM, this means knowing the machine's internal operations and how the Java compiler transforms source code into executable bytecodes. Quite apart from their role in better programs, the mechanics of Java and the JVM are uniquely fascinating. Looking only at the JVM, can you name one other software tool that contains not one but two compilers, three memory reclamation tools, and a specialized performance analyzer that is itself compiled at runtime?

That's why we look into the fundamentals of just-in-time (JIT) compilation in the JVM (page 14), we compare the performance of different garbage collectors (page 20), and we update an article on the JVM's code cache and its effects on performance (page 24). To these, we add deep dives into how Java itself works: how annotations are handled and how you can write your own annotations (page 35), plus examining how the Java Collections Framework was optimized using an unusual technique that is available to you in your code (page 28).

The rest of the issue (see the Table of Contents) shows off a new JVM language, better persistence in Java EE, and how enums work in Java, topped off by our famous language quiz. Enjoy!

ART BY I-HUA CHEN

ANDREW DINN AND
ANDREW HALEY

What Is the JIT Compiler Actually Doing?

How the JIT transforms your code

Most Java developers know that the JVM employs a just-in-time (JIT) compiler to improve the performance of Java programs. However, it's less commonly understood what a JIT compiler actually does and what benefits it provides. In this article, we explain how a JIT compiler functions and why that makes such a difference to performance. We discuss OpenJDK, the open source JDK release, but everything we say applies equally well to Oracle's releases.

Interpretation versus Compilation

Java classes must be compiled to bytecode by the `javac` compiler before they can be executed by the JVM. As explained in a previous article in *Java Magazine* ("How the JVM Locates, Loads, and Runs Libraries"), bytecode encodes the details of classes in a portable, architecture-neutral file format. However, to execute that bytecode you need a JVM (the `java` program) that is built for a specific processor and operating system.

OpenJDK's `java` program provides more than one way of executing Java methods. Initially they are executed by an interpreter that implements the stack-based virtual machine described by *The Java Virtual Machine Specification*. That interpreter provides a full implementation of Java. However, because it uses interpretation, it cannot deliver the best performance.

At a high level, an interpreter cannot perform many optimizations, such as reordering a computation, in order to remove or bypass redundant computations. At the low level, an interpreter cannot make best use of the underlying processor instruction set or the caches and the memory system.

In contrast, a JIT compiler can provide improvements at both these levels. JIT compilers use high-level strategies to transform bytecode into equivalent operations that have the same effect but perform far less computation. These operations are then encoded as low-level, native machine code using the processor instruction set to best effect while ensuring that useful data is retained as far as possible in processor registers or caches, minimizing memory system delays. Also, runtime constants such as cache and heap sizes and the number of available processors can be taken into account when generating code.

As an example of how slow interpretation can be, look at the following program that computes elements of the Fibonacci series:

```
x0 = 1
x1 = 1
xn = xn-1 + xn-2 for n > 1
```

The implementation is in a single Java class:

```
class Fib
{
  public static
  void main(String[] args) {
    int i = Integer.valueOf(args[0]);
    System.out.println(fib(i));
  }

  private static int fib(int i) {
    if (i <= 0) {
      return 0;
    } else if (i == 1) {
      return 1;
    } else {
      return (fib(i - 1) +
              fib(i - 2));
    }
  }
}
```

You can run this using only the interpreter by passing flag `-Xint` to the `java` command:

```
$ time java -Xint Fib 48
512559680

real   18m7.649s
user   18m8.074s
sys    0m0.088s
```

Note that the `time` command is Linux- and UNIX-specific. [Microsoft Windows users can use the ptime utility by Jem Berkes. It is also available in our download area. —*Ed.*]

A JIT-Generated Bytecode Interpreter

Slow as that might seem, it is worth noting that the OpenJDK interpreter is faster than most. One reason is that it is implemented using machine code generated by the JIT compiler. When the JVM starts up, the JIT compiler generates the interpreter as small snippets of native machine code, one for each operation that can appear in bytecode. The interpreter works on one bytecode operation at a time, jumping to the generated code for that operation and then executing the next bytecode.

Each JVM thread has a JVM stack that stores frames, one per method call. Each frame contains a stack known as the *operand stack.* (See *The Java Virtual Machine Specification* for more details.) Many bytecode operations involve popping one or more values off the operand stack; computing a result; and, perhaps, pushing it back on the stack or, alternatively, writing it as the value of a local variable. The processing step might require adding two numbers, fetching a value from an input object's field, or accessing an array element at some given index. Such operations are easily translated into one or two native machine instructions.

Flow-control operations, such as `if` or `while`, update the bytecode pointer, possibly skipping forward to a `then` or `else` branch or backward to a `while` loop condition test. Calls to method operations, such as `invokevirtual`, create a new frame, using the arguments on the operand stack to populate a new locals area, while `return` operations delete a frame and might push a method result onto the caller's operand stack.

Generating the interpreter code at startup has important benefits over writing it in a high-level language such as C++. The instruction sequences are encoded using a low-level assembler, which means they can make better use of the capabilities of the runtime processor than an interpreter written in a high-level language. For example, dedicated machine registers can be used to provide fast reference to commonly used values such as the current Java thread and frame or the bytecode pointer. You can even generate interpreters with different features depending on command-line options.

There are some opportunities for a JITed interpreter to apply optimizations. For example, some pairs of bytecode instructions that are frequently seen together can be merged. One such case occurs when a field of `this` is loaded. The merger requires executing two bytecode instructions, `aload_0` followed by `getfield`. `aload_0` reads the object reference for `this` from the locals area into a register, which is always in local slot zero, and then pushes the value onto the top of the stack. `getfield` pops an object reference off the top of the stack back into the same register, loads a value at some offset from the start address of the object into a second register, and then pushes the loaded value onto the top of the stack. The push and pop in the middle are simply wasted work. In addition, executing each successive instruction requires reading two instruction bytes twice and two jumps to the associated code template for that instruction. The JVM substitutes a pseudo-instruction `fast_agetfield` for the original pair, avoiding the redundant push and pop operations and requiring only one bytecode read and jump. Also, some commonly used library calls—for example, `java.lang.ref.Reference.get()`—have specially coded implementations that enable the interpreter to execute them quickly.

JIT Compilation

To do even better than this optimized interpreter can do, you need a JIT compiler. Even in a JIT compiler, however, not all methods are compiled to optimized machine code. The JVM concentrates on the ones that provide the most benefit.

A JIT compiler must do a lot of work to analyze the bytecode and generate optimized machine code, even for a simple method. Extensive testing of Java code from many sources shows that most methods are only called a very small number of times—and some not at all! So, there is no point bothering to compile all of them. The amount of time saved would be negligible, and the JIT compiler would be better off focusing its time on code that is called frequently. A good JIT compiler tries to speed up execution only for *hot* methods: that is why the OpenJDK JVM was named *HotSpot*. It is often said that 90 percent of the execution time is spent executing 10 percent of the code, so it's ideal to concentrate the optimization efforts on that 10 percent.

You can see the HotSpot VM in action as follows:

```
$ time java -XX:+PrintCompilation Fib 48
   66    1  3 java.lang.String::indexOf (70 bytes)
   67    3  3 java.lang.String::hashCode (55 bytes)
   . . .
   73   10  3 java.lang.String::equals (81 bytes)
   73   11  3 Fib::fib (29 bytes)
   74   12  4 Fib::fib (29 bytes)
   75   11  3 Fib::fib (29 bytes)   made not entrant
512559680

real   0m26.866s
user   0m26.861s
sys    0m0.021s
```

Notice that this compiled code runs more than 50 times faster than interpreting. (The third column in this log shows the optimization level, which we describe shortly.)

Note that the method `String.indexOf` is the first method to be compiled, even though there is no call to that method in the program. This is because starting up Java executes code to set up the runtime environment, and much of it is string processing. Therefore, the first hot methods are encountered in class `String`.

You can see that `fib` is compiled and then recompiled at a higher optimization level. The original compiled code is decommissioned once the new version is in place. So, there seems to be more than one way to compile a method: in fact, there is more than one JIT compiler in OpenJDK.

A Choice of JIT Compilers

The performance improvement provided by compilation comes from executing machine code instead of interpreting bytecode. However, by itself that doesn't necessarily make a big difference. The most significant gains arise from the high- and low-level optimizations that the compiler performs when generating machine code.

OpenJDK includes two JIT compilers, often known as the *client* and *server* compilers. Originally you had to pick one or the other on the `java` command line (using either the `-client` or `-server` options). In recent JDK releases, the default configuration is to use them both in what is called *tiered compilation* mode. You can see tiered compilation in action in the previous trace. Method `fib` is first compiled at Level 3, that is, using the client compiler but including code to count method calls and paths taken at branches. `fib` is then recompiled at Level 4, which uses the server compiler.

The client JIT compiler was designed for desktop applications that typically run only for a short time, possibly only for a few seconds or, perhaps, a few minutes. In contrast, the server JIT compiler was intended for applications that run for hours, days, or even months. Both compilers optimize, but they make very different trade-offs. Essentially, the client JIT compiler performs less optimization than the server JIT, producing slower compiled code but generating it more quickly.

The trade-off is easy to see. For example, in the OpenJDK source code from the OpenJDK repository, the Java library source is in the subdirectory `jdk/src/share/classes`. We ran `javadoc` with only the client compiler, as shown below, where the input file `jdkfiles` lists all Java source files found below `src/share/classes`:

```
$ time javadoc \
-quiet -J-XX:TieredStopAtLevel=2 @jdkfiles
real    4m41.775s
user    5m31.390s
sys     0m1.862s
```

Command-line option `-J` is used to pass an argument to the underlying JVM. The option `-XX:TieredStopAtLevel=2` asks the JVM to execute only at Level 1 (interpreted) or Level 2 (client compiler without profiling).

To run the JVM using only the server compiler, you would pass the option `-XX:-TieredCompilation`, which switches off tiered compilation. It generates these timings:

```
$ time javadoc \
-quiet -J-XX-TieredCompilation @jdkfiles
real    3m30.083s
user    4m50.410s
sys     0m1.880s
```

Going to the server compiler has dropped the elapsed time from 4 minutes and 40 seconds to 3 minutes and 30 seconds, or a reduction of 25 percent. It is interesting to look at the change in user time, which is the amount of CPU time used across all of the cores. In the first run, 50 seconds more CPU time was used than real (elapsed) time. In the second run, there was 1 minute and 20 seconds of extra CPU time, a 60 percent increase. So, 60 percent more JIT compiler time cut 25 percent off the total execution time.

JIT compilation is done in background threads, normally on an otherwise idle CPU. So, compilation doesn't slow down an application by stealing the CPU. The usefulness of a JIT compiler depends on two things: the speed of the code it generates and how fast it delivers that code. Delivery time is important because execution switches to compiled code only after

- The method is called enough times to be queued for compilation
- The compiler dequeues the method and generates the compiled code

The server compiler produces faster code, but that faster code has some catching up to do. Because each server-compiled method arrives later than the corresponding client-compiled method, methods that are hot when scheduled for compilation might have gone cold by the time the compiled code is generated. The program might even exit before the server compiler has finished.

Therefore, for a server application, if you have to choose between the JITs, the server compiler is almost certainly a better choice than the client compiler. It is well worth sacrificing a bit of lost time getting up to speed during server startup if it ensures that code that runs for weeks is as highly optimized as possible.

However, OpenJDK provides an even better option that, for most applications, provides the best of both worlds: the default tiered compilation configuration we mentioned earlier. *Tiered* means that hot methods are first compiled using the client JIT compiler. These client-compiled methods count how often they are executed, and if they remain hot, they are recompiled using the server JIT compiler. Tiered compilation gives good startup speed while ensuring that code that remains hot is fully optimized. It also avoids wasting time heavily optimizing code that is called a lot at program startup but then quickly goes cold.

When run with tiered compilation, the `javadoc` example doesn't show much difference compared with the server compiler numbers:

```
$ time javadoc \
-quiet -J-XX+TieredCompilation @jdkfiles
real   3m31.275s
user   5m23.726s
sys    0m2.093s
```

You can see that there is a lot more user time (1 minute 50 seconds) spent on compilation but the elapsed time does not really improve much. That's because almost all the code in the `javadoc` application that gets client-compiled is executed frequently enough that it eventually gets server-compiled. So, the benefit of delivering JITed code earlier is lost in the overhead of running that slower code for longer.

For most real-life server applications, tiered compilation rarely costs much and is frequently a much better bet than plain server compilation, because the applications start up considerably faster. Tiered compilation also helps interactive applications, again because they start up quickly and then get faster as the server compiler does its work. You might have noticed this effect when running applications such as NetBeans.

Some Optimizations Are Possible Only with a JIT

You might ask, what is the point of using a JIT compiler when it adds the overhead of compiling methods at runtime? Why not just compile all the code in advance to get the best performance, as is done with native languages such as C++? (This is called *ahead-of-time compilation*, or AOT.) That would be possible if all the code were available at compile time but, of course, Java is a dynamic language that is able to load code from the classpath or even via the network. If there were no JIT compiler, dynamically loaded code would need to be interpreted, which would be highly unsatisfactory.

However, that is not the only reason for using a JIT compiler. The most important benefit of runtime compilation is that certain optimizations become possible that are not available to AOT compilers. It is not widely understood or acknowledged, but the OpenJDK JIT compiler can generate code for a whole range of programs faster than, say, a C++ compiler can do for an equivalent program.

Conclusion

If you are interested in seeing what the JVM does with your application code, try running with the following flags enabled:

```
java -XX:+PrintCompilation \
    -XX:+UnlockDiagnosticVMOptions \
    -XX:+PrintInlining MyMainClass arg1 … argN
```

These options enable you to observe inlining decisions being made as your methods are compiled. Over a long enough time and with a varying data set, you might see your methods being deoptimized and recompiled as application-phase changes drive your code down previously cold paths. You might see increasingly larger inline trees being submitted for compilation as call counts accumulate from the bottom up.

If you are really adventurous, you can download the OpenJDK source code and build the disassembler library (look for the `hsdis` subdirectory in the `hotspot` source tree), allowing you to print the generated native machine code with the `-XX:+PrintAssembly` option. You might even build a debug version of OpenJDK, which provides a host of other `Print` and `Trace` options to expose a lot more information about what the JIT compiler is doing on your behalf.

It has often been observed that the most successful technology is invisible: it works so well that you don't know it's there. Almost all of the time, the Java HotSpot VM is like that. The interpreter, the JIT compilers, the garbage collectors, and the runtime system all work together so smoothly and quickly that you don't notice that they're there. But sometimes, knowing what is really going on can give you clear benefits. `</article>`

Andrew Dinn is a member of Red Hat's OpenJDK team and also leads the JBoss project Byteman.

Andrew Haley is technical lead of Red Hat's Java team. He has been programming professionally for more than 30 years and using Java for almost as long as it has existed.

learn more

"Introduction to JIT Compilation in Java HotSpot VM"

BULGARIAN JAVA USER GROUP

Bulgaria is a popular spot for startups and outsourcing. The Bulgarian Java User Group (JUG) was founded in September 2007 along with a mailing list for discussion of Java-related issues. Five members set up a leadership board in 2013 and began regular meetups and sessions once or twice a month. The JUG took part in Adopt OpenJDK activities and soon joined the Java Community Process program. As a member, the JUG pushed for some changes to OpenJDK and held a few hackathons.

In 2015, the group started organizing a community conference called jPrime, which attracted 400 attendees and 20 sponsoring companies. The conference is considered one of the major Java events in the region.

The group also started up jProfessionals, a series of free, one-day miniconferences. The first jProfessionals meeting was held in November 2015, and the featured speaker was Kohsuke Kawaguchi, the creator of Jenkins CI.

The Bulgarian JUG organizes regular events with local and foreign presenters, including Java Champion David Blevins, who spoke about the TomEE application server. Monthly events include hands-on labs.

During the summer, when there are no meetups, members have started weekend code retreats. The topic in 2015 was developing the JBoss Forge add-on for Spring Boot.

The plan for 2016 is to put on even more events.

CHRISTINE H. FLOOD

The New Garbage Collectors in OpenJDK

The upcoming G1 and Shenandoah garbage collectors

Life in the world of OpenJDK garbage collection (GC) is getting exciting. Not only is there a new default JDK 9 garbage collector, called G1, but there is also a new alternative, Shenandoah, from Red Hat. In this article, I discuss the differences between the current parallel collector and G1, and I examine what Shenandoah brings to the table.

What Is GC?

GC is an automated way to reclaim for reuse memory that is no longer in use. Unlike other languages in which objects are allocated and destroyed manually, with GC, programmers don't need to pick up and examine each object to decide whether it is needed. Instead, the omniscient GC housekeeper process works behind the scenes quietly discarding objects that are no longer useful and tidying up what's left. This decluttering leads to an efficient program.

The JVM organizes program data into objects. Objects contain fields (data) in a managed address space called a *heap*. Imagine the Java class below, which represents a simple binary tree node.

```
class TreeNode {
    public TreeNode left, right;
    public int data;
    TreeNode(TreeNode l,  TreeNode r, int d) {
        left = l; right = r; data = d;
    }
    public void setLeft(TreeNode l) { left = l;}
    public void setRight(TreeNode r) {right = r;}
}
```

Now imagine the following operations performed on this class.

```
TreeNode left = new TreeNode(null, null, 13);
TreeNode right = new TreeNode(null, null, 19);
TreeNode root = new TreeNode(left, right, 17);
```

Here, I've created a binary tree with a root of 17, a left subnode of 13, and a right subnode of 19 (see **Figure 1**).

Suppose I then replace the right subnode, leaving subnode 19 as unconnected garbage:

```
root.setRight(new TreeNode(null, null, 21));
```

This results in the situation shown in **Figure 2**.

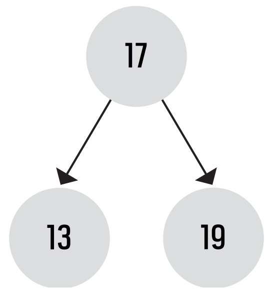


Figure 1. A three-node tree

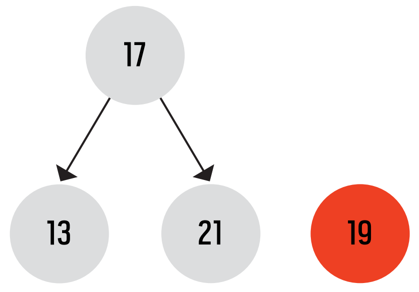


Figure 2. The same tree with one subnode replaced

As you can imagine, in the process of constructing and manipulating data structures, the heap will start to look like Figure 3.

Compacting the data means changing its address in memory. The Java program expects to find an object at a particular address. If the garbage collector moves the object, the Java program needs to know the new location. The easiest way to do this is to stop all the Java threads, compact all the objects, update all the references to the old addresses to now point to the new addresses, and resume the Java program. However, this approach can lead to long periods (called *GC pause times*) when the Java threads aren't running.

Java programmers aren't happy when their applications aren't running. There are two popular strategies for decreasing GC pause times. The GC literature refers to them as *concurrent algorithms* (doing work while the program is running) and *parallel algorithms* (employing more threads to get the work done faster while the Java threads are stopped). The current OpenJDK default garbage collector (which can be manually specified on the command line with `-XX:+UseParallelGC`) adopts the parallel strategy. It uses many GC threads to get impressive throughput.

Shenandoah compacts the data concurrently. As a consequence, Shenandoah doesn't need to limit the number of regions it collects in order to minimize application pause times.

Parallel Garbage Collector

The parallel garbage collector segregates objects into two regions—*young* and *old*—according to how many GC cycles they have survived. Young objects are initially allocated in the young region, and the compaction step keeps them in that region until they have survived a certain number of young collections. If they live long enough, they are promoted to the old generation. The theory is that rather than pausing to collect the entire heap, which would take too long, you can collect just the part of the heap that is likely to contain short-lived objects. Eventually it will become necessary to collect the older objects as well.

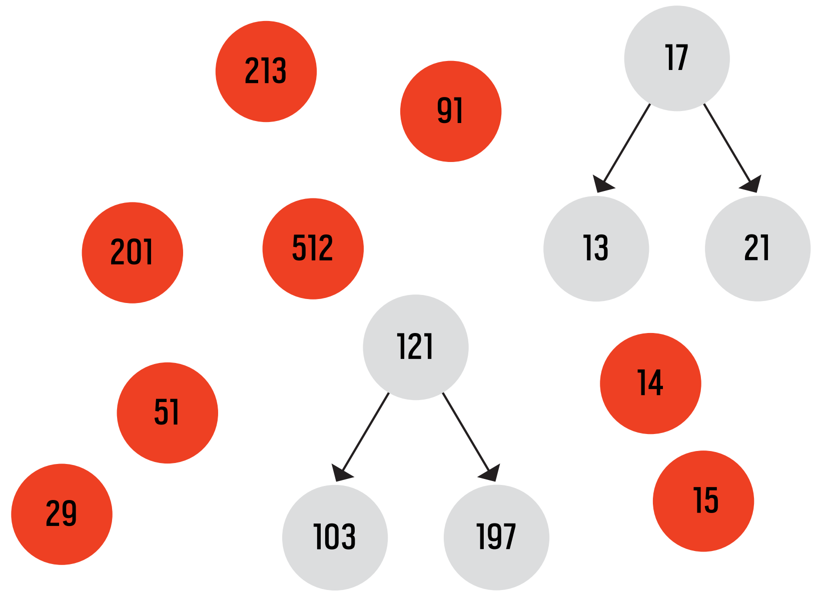


Figure 3. A heap with many unused data items in it

In order to collect just the younger objects, the garbage collector needs to know which objects in the old generation reference objects in the young generation. The old objects need to be updated to reference the new locations for the new objects. The JVM does this by maintaining a summarization data structure called the *card table*. Whenever a reference is written into an old-generation object, the card table is marked so that during the next young GC cycle, the JVM can scan this card looking for old-to-young references. With these references known, the parallel garbage collector is able to identify which objects to cull and which references to update. It uses multiple GC threads to get the work done faster while it has paused the program.

Garbage-First Garbage Collector

The new JDK garbage collector—named G1—uses both parallel and concurrent threads. It uses concurrent threads to scan the live objects while the Java program is running. It uses parallel threads to copy objects quickly and keep pause times low.

G1 divides the heap into many regions. A region might be either an old region or a young region at any time during the program run. The young regions must be collected at every GC pause, but G1 has the flexibility to collect as many or as few old regions as it predicts it can collect within the user-specified pause-time goal. This flexibility allows G1 to focus the old-object GC work on the areas of the heap that have the most garbage. It also enables G1 to tune collection pause times based on user-specified pause times.

As shown in **Figure 4**, G1 will freely compact objects into new regions.

G1 knows how much data is live in each region and the approximate time it takes to copy that live data. If the user is interested in minimal pause times, G1 can choose to evacuate only a few regions. If the user is not worried about pause times or has stated a fairly large pause-time goal, G1 might choose to include more regions.

The key difficulty with Shenandoah's concurrent copying is that the GC threads doing the copying work and the Java threads accessing the heap need to agree on an object's address.

G1 must maintain a card table data structure so that it can collect only young regions. It also must maintain a record for each old region that other old regions have references to. This data structure is called an *into remembered set*.

The downside of specifying small pause times is that G1 might not be able to keep up with the program allocation rate, in which case it will eventually give up and fall back to a full stop-the-world GC mode. This means that both the scanning and the copying work are done while the Java threads are stopped. Note that if the GC can't meet the pause-time goal with partial collections, then a full GC is guaranteed to exceed the allocated time.

In sum, G1 is a good overall collector that balances throughput and pause-time constraints.

Shenandoah Garbage Collector

The Shenandoah garbage collector is an OpenJDK project that is not yet part of the OpenJDK distribution. It uses the same region-based heap layout as G1 and employs the same concurrent scanning threads to calculate the amount of live data in each region. It differs in the way it handles the compaction stage.

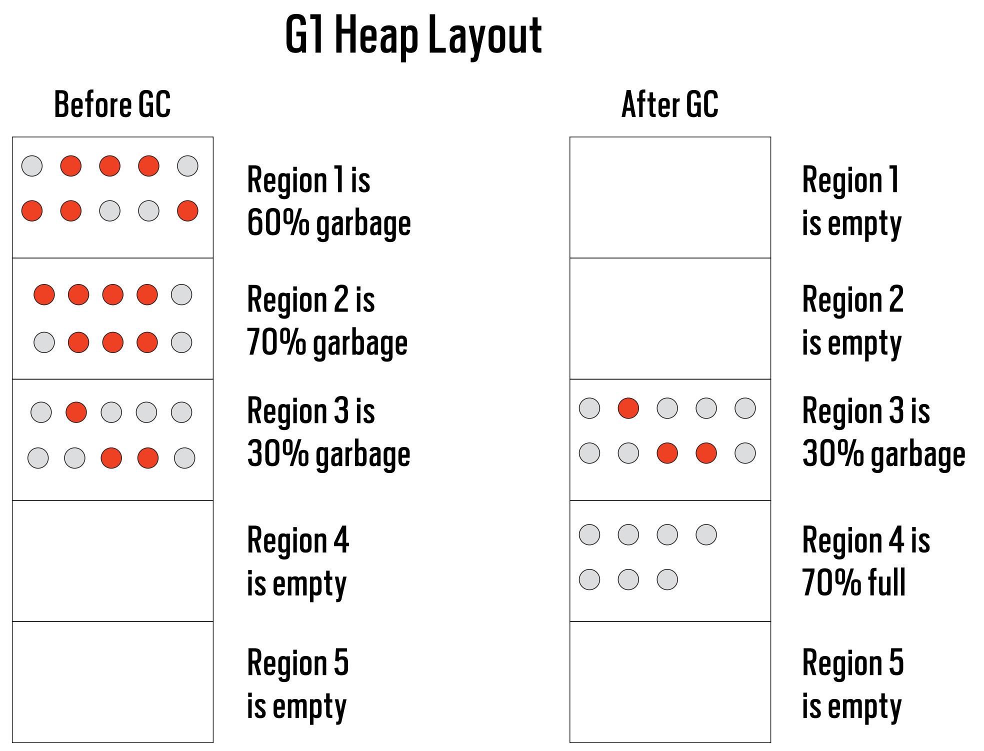


Figure 4. Before and after a G1 run. Regions 1 and 2 are compacted into region 4. New objects may be allocated to fill region 4. Region 3 is untouched because there would be too much copying work (70 percent) for too little space reclamation (30 percent).

Shenandoah compacts the data concurrently. (The sharp-eyed among you will have noticed that this means it might need to move objects around while the application is trying to read them or write to them; don't worry—I'll come to that in a second.) As a consequence, Shenandoah doesn't need to limit the number of regions it collects in order to minimize application pause times. Instead it picks all the most fruitful regions—that is, regions that have very few live objects or, conversely, a lot of dead space. The only steps that introduce pauses are those associated with certain bookkeeping tasks performed at the beginning and end of scanning.

The key difficulty with Shenandoah's concurrent copying is that the GC threads doing the copying work and the Java threads accessing the heap need to agree on an object's address. This address might be stored in several places, and the update to the address must appear to happen simultaneously. Like most thorny problems in computer science, the solution is to add a level of indirection.

Objects are allocated with extra space for an indirection pointer. When the Java threads access the object, they first read the indirection pointer to see whether the object has moved. When the garbage collector moves an object, it updates the indirection pointer to point to the new location. New objects are allocated with an indirection pointer that points to themselves. Only when an object is copied during GC will the indirection pointer point to somewhere else.

This indirection pointer is not free. It has a cost in both space and time to read the pointer and find the current location of the object. These costs are less than you might think. Spacewise, Shenandoah does not need the off-heap data structures used to support partial collections like the card table and the into remembered sets. Timewise, there are various strategies to eliminate read barriers. The optimizing JIT compiler can realize that the program is accessing an immutable field, such as an array size. It's correct in those cases to read either the old or the new copy of the object so no indirection read is required. In addition, if the Java program reads multiple fields from the same object the JIT may recognize this and remove the subsequent reads of the forwarding pointer.

If the Java program writes to an object that Shenandoah is copying, a race condition occurs. This is solved by having the Java threads cooperate with the GC threads. If the Java threads are about to write to an object that has been targeted for copying, the Java thread will first copy the object to its own allocation area, check to see that it was the first to copy the object, and then perform the write. If the GC thread copied the object first, then the Java thread can unwind its allocation and use the GC copy.

Shenandoah eliminates the need to pause during the copying of live objects, thus providing much shorter pause times.

Conclusion

If you are interested in the best end-to-end throughput, you will probably want to use the parallel garbage collector that currently ships in the JDK. If you want a good compromise between pause times and throughput, the new G1 garbage collector will work well for you. Shenandoah will be an attractive option for response-time-critical applications running with large (more than 20 GB) heaps such as financial trading, ecommerce, and other interactive applications in which the user would be irritated by noticeable GC delays. </article>

Christine H. Flood is a principal software engineer for the Java platform at Red Hat, where she works on Shenandoah.

learn more

A 2014 presentation about Shenandoah by the author

Oracle tutorial on garbage collection in the JVM

BEN EVANS

Understanding the Java HotSpot VM Code Cache

Learn to detect and mitigate a full code cache.

Java HotSpot VM has an advanced just-in-time (JIT) compiler that enables Java HotSpot VM to produce very highly optimized machine code for any platform that Java HotSpot VM runs on.

In this article, I examine an important aspect of Java HotSpot VM's JIT compiler: the code cache. Understanding the code cache provides insight into a range of performance issues that are otherwise difficult to track down.

Note: An article in this issue, "What Is the JIT Compiler Actually Doing?," and a previous article in *Java Magazine*, "Introduction to JIT Compilation in Java HotSpot VM," [PDF] discuss introductory Java HotSpot VM and JIT compiler topics.

To start our journey toward the JIT compiler and code cache, let's begin by considering the lifecycle of a Java method.

Lifecycle of a Java Method

The smallest unit of new code that the Java platform will load and link into a running program is a class. This means that when a new method is being onboarded, it must go through the class-loading process (as part of the class that contains it).

The class-loading process acts as a *pinch point*: a place where a lot of the Java platform's security checks are concentrated. The lifecycle of a Java method, therefore, starts with the class-loading process that brings a new class into the running JVM.

PHOTOGRAPH BY JOHN BLYTHE

Class Loading

Class loading starts with a stream of bytes (often read from disk) that should be in the class file format. If the byte stream fits into the expected format, the class loader can attempt to link it.

The linking process has several phases, of which the first—and most important—is verification. This is the phase in which the JVM confirms that the new class file does not attempt to violate Java's robust programming model.

During the verification phase, several security constraints are checked. For example, it is verified that

- Methods respect access control keywords
- Methods are called with correct static types
- Variables are assigned only suitably typed values
- Variables are properly initialized before use

The bytecode of methods is also extensively checked. A key point here is that the JVM is a stack machine.

This choice was a deliberate one—it is much easier to prove security (and other) properties on a stack machine than with a register-based machine. This means that most of the checks to be made on bytecode can be done economically via static analysis at class-loading time, which greatly reduces the chance of harmful code ever making it into a live JVM.

For example, the stack state can be deduced at every point in a method without needing to keep track of the contents of registers.

Note that for performance reasons, JDK classes (from rt.jar) are not checked. They are loaded by the primordial class loader, which doesn't do comprehensive security checks.

This use of class loading as the opportunity to verify bytecode slows down the class-loading process. However, the payoff is that it significantly speeds up runtime, because checks can be done once and then omitted when the code is actually run.

How Java HotSpot VM Implements Class Loading

The key method that is used to turn a stream of bytes into a class object is the Java method `ClassLoader::defineClass()`. This method delegates to a native method, `ClassLoader::defineClass1()`, which does some basic checks and string conversion and then calls a C function called `JVM_DefineClassWithSource()`.

As we might expect, this is an entry point into the JVM, and it provides access into the C++ code of Java HotSpot VM. Java HotSpot VM uses the `SystemDictionary` to load a new class via the `parseClassFile()` method of `ClassFileParser`.

Once class loading has been completed, the bytecode of the method is placed inside a C++ object (`methodOop`) for the bytecode interpreter to use.

This is sometimes called the *method cache*, although the bytecode is actually held inline in the `methodOop` for performance reasons.

How Do Methods Get Compiled?

Java HotSpot VM maintains a large number of performance and tracing counters in the bytecode interpreter. These counters trigger the compilation of methods once the methods have been run 10,000 times (for the server compiler).

The code that is output from the compiler is machine code (specialized for the specific operating system and CPU in use). It is placed into a central place—the `CodeCache` (a C++ object)—which is a heap-like structure for holding `CodeBlob` instances (which are the compiled representations of method code).

With the code blobs in the code cache, the running system is then updated to use the new compiled code rather than interpreted mode (this update process, which involves updating pointers, is sometimes called *pointer swizzling*).

PrintCompilation

One of the simplest flags that can be used to control the JIT compilation subsystem is `-XX:+PrintCompilation`. This switch tells the JIT threads to add compilation messages to the standard log. `PrintCompilation` is explained further in the second article I linked to earlier.

Deoptimization

Java HotSpot VM's server mode uses optimizations that it can't always prove hold true. It protects these optimizations with sanity checks (often called *guard conditions*), and if a check fails, Java HotSpot VM will deoptimize the code that was based on that assumption.

It's common for Java HotSpot VM to then reconsider and try an alternative optimization. This means that the same method might be deoptimized and recompiled several times.

We can see deoptimization events in the `PrintCompilation` log; they show up as lines such as "made not entrant" and "made zombie."

These lines mean that a particular method, which had been compiled to a code blob, has now been deoptimized. This usually (but not always) happens because a new class was loaded and invalidated an assumption made by Java HotSpot VM.

What Happens as the Program Warms Up?

After a Java program starts up and goes through its initialization phases, it will normally get into normal operation and the hot paths of code will start to develop.

If you do multiple runs with the `PrintCompilation` switch on and collect the logs of which methods were compiled, a pattern emerges:

- Compilation usually eventually stops.
- The number of compiled methods stabilizes.
- The set of compiled methods on the same platform for the same test inputs is usually fairly consistent.
- The exact details of which methods get compiled depend on the exact JVM, operating system platform, and CPU in use.

Note: The compiled code for a given method is not guaranteed to be roughly the same size across platforms.

Similar size is the usual pattern, but there are cases in which the picture can be different from this; you should always check. One good way to do this is with Java VisualVM, which shows the general shape of the class-loading curve in its Classes section (the lower left panel of **Figure 1**).

What Happens if the Code Cache Fills?

In short, compilation has to stop. This is because once a code blob is compiled, usually only deoptimization can remove it from the code cache.

Code cache space is reclaimed by flushing the "zombie" code blobs from the code cache. (Over time, any "not entrant" blobs turn into zombies.)

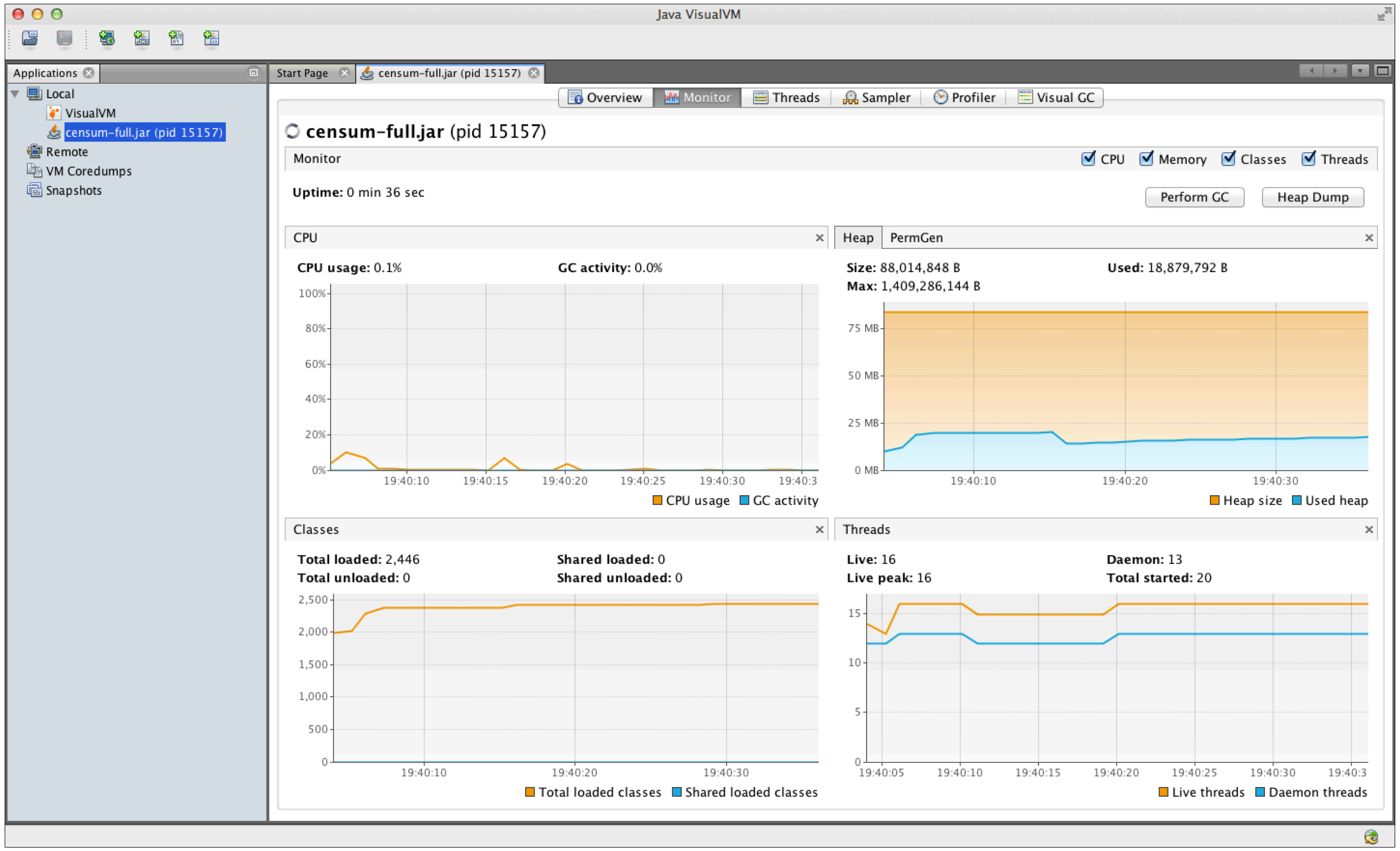
In JDK versions after Java 7 Update 4, there is an additional form of code cache flushing: *speculative flushing*. In this approach, the older methods are marked as being potentially eligible for flushing and they are disconnected from the `methodOop` that created them. If the VM needs to call the compiled method, the method is relinked back to its `methodOop` and survives being flushed.

However, if the method is not called again within a certain time frame, the `methodOop` is reverted to interpreted mode, and the code blob is eligible for being flushed.

What Happens During Startup?

To see why application startup time could be problematic for the code cache, let's consider an imaginary Spring application.

Spring applications start up using the `Bootstrap` class, which locates an XML



Uptime: 0 min 36 sec

CPU usage: 0.1% GC activity: 0.0%

Size: 88,014,848 B Used: 18,879,792 B
Max: 1,409,286,144 B

Total loaded: 2,446 Shared loaded: 0
Total unloaded: 0 Shared unloaded: 0

Live: 16 Daemon: 13
Live peak: 16 Total started: 20

Figure 1. Java VisualVM showing class-loading data

file detailing the instances to be created and wired up (and defines the classes to be loaded).

This means that Spring applications go through two phases of class loading: first, the phase of loading the classes needed to start the bootstrapping, and then a second phase that occurs when the application classes are typically loaded.

From the point of view of JIT compilation, this is important because the Spring framework uses reflection and other techniques to discover which classes to load and instantiate. These framework methods are called heavily during application startup but then are never touched again after that.

If a Spring framework method is run enough to be compiled, it is going to be of minimal use to the application. It will marginally improve application startup time, but at the price of using up a scarce resource. If enough framework methods are compiled, they can use up the entire code cache, leaving no room for the application methods that we actually want to be compiled.

To solve this problem, the JVM uses a system in which the values of counters decay over time. In their simplest form, the decay reduces the invocation counts for methods by 50 percent every 30 seconds.

This means that if methods are used only at startup, their invocation counts will, within a few minutes, have decayed down to effectively zero. This prevents the rarely used Spring framework methods from using valuable code cache space.

Which Switches Control Compilation and the Code Cache?

The following switches control compilation and the code cache:

- `-XX:+PrintCompilation` shows log entries for compilation and deoptimization events.
- `-XX:CompileThreshold=n` changes the number of times a method must be called before being compiled.
- `-XX:ReservedCodeCacheSize=YYm` sets the overall size of the code cache to be used.
- `-XX:+UseCodeCacheFlushing` allows a JVM to flush little-used code blobs (this is on by default in Java 7 Update 4 and later).

How Do We Fix Applications Suffering from a Full Code Cache?

To identify a full code cache and resolve it, first make sure the cache is a limitation. That is always true in the event a "compilation halted" warning is issued. You can check whether the size is too small (and remediate the problem) using these steps.

1. Use `-XX:+PrintCompilation` to output the methods that are actually being compiled.
2. Wait until this reaches steady state.
3. Repeat a few runs. Check that the results set is stable.
4. Try increasing the size of code cache (doubling is often a good first step) using `-XX:ReservedCodeCacheSize`. If more methods are now seen to be compiled, you can be sure that the original code cache was too small.
5. Retest overall performance to ensure that increasing the code cache size hasn't harmed some other aspect of application performance.

Optimizations such as this have an important empirical aspect: once you make a change, you must measure its results carefully. As this article demonstrates, the first step is understanding what the JVM is doing so that you know what changes to try.

Ben Evans helps to run the London Java Community and represents the user community on the JCP Executive Committee.

learn more

Oracle's JVM Specification for Java SE 8

MIKE DUIGOU

For Faster Java Collections, Make Them Lazy

How adding lazy operations to ArrayList and HashMap improved performance and reduced memory usage

The Java core libraries community has been working hard to improve the Java Collections Framework by making it lazier. *Laziness* in software is an architectural or systematic approach that defers producing a result until it can be definitively determined that the result is needed. Many operations can be decomposed into a collection of suboperations. Laziness delays performing suboperations until the results of those operations are needed to complete some other suboperation or the overall operation.

A nonlazy approach to completing any operation is to perform the entire sequence of suboperations and then combine their results to produce the final result. The more efficient lazy approach is to begin by combining the results of suboperations. Whenever you discover a needed missing result from an unsolved suboperation, you perform that suboperation to determine the subresult. Initially you start with no computed subresults and accumulate results by completing suboperations, enabling additional suboperations to be completed and culminating in a final result for the whole operation. Laziness successfully saves time if, when you have the result of an operation, there were suboperations whose results were never determined because those values were not needed to determine the final operation result.

Anytime that the result of an operation is potentially or likely not going to be needed as part of a final result, it makes sense to defer that operation until it is determined that the result is actually needed. The most common example of laziness occurs in expression evaluation. Consider the following code:

```
int x = 5;
int y = 3;
if (x < 2 && y < 7) {
 ...
```

The simplest way to evaluate this expression would be to evaluate each term and combine the terms:

```
5 < 2 => FALSE
3 < 7 => TRUE
FALSE && TRUE => FALSE
```

Notice that if the first term evaluates to false, then the result of the entire expression will always be false. Therefore, we need not bother evaluating the second term unless the first term evaluates as true.

For Java programs, the *Java Language Specification* specifies that terms of an expression are evaluated left to right and any terms not needed for the result will not be evaluated at all. This often saves computation. It is also very useful:

```
if (foo != null && foo.bar() == 3) {
```

In this example, calling the `bar()` method requires that foo be nonnull. If foo is null then attempting to evaluate the second term of this expression would cause a NullPointer Exception. The Java lazy evaluation rules ensure that the second term is evaluated only when the first term is true. Logical AND (&&) terms can be thought of as equivalent to nesting, so

```
if (foo && bar && baz) {
```

is equivalent to

```
if (foo)
 if (bar)
   if (baz)
```

This rewriting as nested conditionals also makes it clearer why the unneeded terms are not evaluated. Lazy evaluation also applies to logical OR (||) expressions, such as

```
if (true || something) {
```

The `something` term of this conditional expression is never evaluated because the value of the expression can be determined before it is evaluated.

These examples of laziness in expression evaluation are useful in writing efficient and, probably just as importantly, concise program logic. Other forms of laziness are just as beneficial, but the connection between decisions made in program flow and the benefit received usually isn't as direct or immediate.

If the result of a calculation is occasionally or frequently discarded without being used, then it makes sense to avoid using the resources required to produce it until it is necessary. The most obvious saved resource is CPU cycles, but laziness can also save memory in avoided allocations and system resources in avoiding unnecessary files, sockets, threads, database connections, and more. Depending on the situation, these savings can be substantial.

In a smaller number of applications, I found that laziness provided up to a 20 percent reduction in memory usage and a similar reduction in memory churn.

Implementing laziness can be a critical optimization strategy in improving system performance. It improves performance by avoiding unnecessary work rather than improving the efficiency of performing the work. Laziness is akin to reducing the number of database queries an application makes by 30 percent as opposed to improving the performance of database queries in the same app by 3 percent. Spending your effort on the former, if it is feasible, is much more effective.

The Challenge of Lazy Collections

The implementation of laziness in the Java Collections Framework, which is already quite well optimized, came about as a result of analysis of application behavior. The Oracle Performance Scalability and Reliability (PSR) team evaluated the performance of some Oracle frameworks and the applications that ran on those frameworks. The PSR team found that it was quite common for both the application and the middleware to allocate `ArrayList` and `HashMap` instances that were then never used in the life of the object that contained them. About 2 percent of all allocated `ArrayList` and `HashMap` instances never received any elements. Further analysis found that the collections were used in some cases, but weren't always needed. Some work by the PSR team was done to see if refactoring the application to handle the cases where the collections were needed and where they weren't could be handled by separate classes with a common base class and with the sometimes-unused collections defined in one of the subclasses. This approach turned out to not be viable because a lot of the cases were similar to the following representative example:

```
public abstract class RestRequest {
   protected final Map<String,String> httpHeaders =
       new HashMap<>();
   protected final
       Map<String,List<String>> httpParams =
           new HashMap<>();
   protected final Set<Cookie> httpCookies =
       new HashSet<>();
```

`RestRequest` is a made-up example class that would be used in an application handling HTTP REST queries. REST is a common approach to building APIs using HTTP for communications. In this example, each `RestRequest` object is a specifically formatted HTTP request that represents a call to a REST API provided by the host application. Each `RestRequest` instance needs to present the important aspects of an HTTP message to the application receiving the request. This includes the HTTP headers (`httpHeaders`), HTTP parameters from either query string or form data (`httpParams`), and HTTP cookies (`httpCookies`). Each of these types of HTTP features may be present in any HTTP message, but the exact usage is determined by the individual application providing the REST API and the client applications using the REST API.

Because the usage of HTTP features for any given request is indeterminate, including the data structures in `RestRequest` for each potential feature is problematic. HTTP headers, parameters, and cookies are common and required parts of the HTTP protocol, but applications aren't obliged to use all of these features and may even choose to use none of them. Some REST APIs may use HTTP parameters where others might use cookies and headers. One possible approach to handling optional features would be to provide many variants of the `RestRequest` class to express all the possible combinations of HTTP features that might be used. This would be both annoying and inconvenient to use, though. Even when it could be determined that a particular HTTP feature will be used for a particular type of request, it is common for that feature to be used only on a fraction of the requests. Consider that authenticated users might use cookies, whereas unauthenticated users of the same request would not. Perhaps a majority of requests are from unauthenticated users.

It makes program logic much simpler to have a single `RestRequest` class with the `httpParams` field always available and initialized whether it is used or not. (I'll get back to this.)

Because the framework or application couldn't be refactored to eliminate the indispensable but frequently unused `httpParams` and `httpCookies` fields, alternatives were needed.

The overall goal was to improve application performance by avoiding the cost of having a field like `httpParams` in a class unless the field was actually used. One solution would have been to lazily initialize the `httpParams` field in the `RestRequest` class—that is, create the `HashMap` only when HTTP parameters were found to be present. This would have required the addition of guard checks around all uses of the `httpParams`:

```
if( httpParams != null )
```

But because the `RestRequest` class is designed to be extended, all subclasses that extend `RestRequest` would need to have similar checks on their use of `httpParams`. Because there was a lot of existing code without these checks, it was unreasonable to suggest that `RestRequest` might suddenly stop consistently initializing the field.

Allowing httpParams to sometimes be null would also have made the logic of methods more complicated with all of the added guard checks that would require. If the same approach were used on many fields in RestRequest, the logic of RestRequest methods and subclasses would be overly focused on repeated checks to determine which features of RestRequest were in use. Over time, inevitably, mistakes would creep in or cases would be forgotten when other parts of RestRequest changed. One missing guard check on a field containing null could ruin your whole day!

Java 8 also introduced a significant new lazy implementation, the Streams API. **This library utilizes laziness as a core principle.**

One solution—which is often useful, but not in this case—would have been to leave unchanged the declaration portion of httpParams,

```
protected final Map<String,List<String>> httpParams;
```

and move the initialization to constructors. In the constructors, if it could be determined that no HTTP parameters were present in a request, the httpParams field could be initialized to Collections.emptyMap() rather than creating a unique HashMap instance for every request. (Collections.emptyMap() along with emptyList() and emptySet()—these and other similar utility methods in the Collections utility class are an efficient way to provide an empty collection. These empty collections are often used rather than creating a unique instance for an empty item. They are frequently better than returning null because they require no more space and, by not returning null, the checks for a null result that would be required can be eliminated.)

After first pursuing adding laziness within the frameworks and applications by avoiding creating HashMap and ArrayList instances, it became clear that the most effective approach was to implement the laziness inside the Java Collections Framework itself.

Updating Java Collections

Making modifications to fundamental Java classes such as ArrayList and HashMap is serious business. There are millions of programmers and billions of lines of code using these classes, and both the programmers and the programs expect that Java will provide reliable, consistent behavior and performance from version to version. The Java Collections Framework is a contract with developers and programs to provide specified behavior. It is essentially impossible to redefine the functionality of JDK classes—that is, to change the contract—in Java updates or even major releases. Some small refinements to the API contract are possible, but most improvements available to the Java Collections Framework are internal changes. Even internal changes must be considered carefully to ensure that they do not have unwanted side effects or cause unexpected behavior changes.

Earlier, I said it would be difficult to use a class such as the RestRequest example if some of the fields might be null. Potentially null public or protected fields require additional work for anyone accessing them. Every dereference of the field must be guarded by a check that the field is not null. Failure to consistently check for null is a common error in programs that have nullable fields in base classes. It is often recommended not to allow protected or public fields to be null. Handling potentially null fields is slightly more manageable when the field is private. This is because all references to the field are in a single file and it is much easier to reason about the possible values of the field in all object states.

Both ArrayList and HashMap use a *package private* array field as their core data structure for storing elements or map entries. Other than the ArrayList or HashMap object itself, this array is the only other memory allocated by ArrayList and for empty or nearly empty HashMaps, the array is the biggest memory allocation. The heart of the laziness upgrade to both classes was to add guard checks for the array field being null.

The primary benefit of the laziness added to ArrayList and HashMap is that it delays and potentially avoids allocating the array until the moment the first element is placed in the array or map. Allocation of the memory used for the backing array is a significant cost for some applications. For other applications, particularly where the unused collection is very short-lived and none of the memory was known to be allocated after the allocating method was complete, the benefit is in saved computation from initializing the array.

Because the field reference involved is for an array, the JVM was already required to check that the array was non-null before either indexing into the array or determining the length of the array. The added guard checks were explicit versions of implicit null checks that were already happening. This meant that adding the guard checks caused no performance penalty.

A second concern was that having the additional guard checks and allocation logic in various methods would change how HotSpot would choose to inline the methods or, more important, not inline the methods, which could potentially undermine performance. Inlining is an optimization used by HotSpot for small methods. When HotSpot is compiling code that invokes a short or simple method, it will often replace the method call with the actual code of the invoked method. There is a size limit on which methods HotSpot will inline. Examination found that we were not near the inlining limit boundary on any of the critical methods modified by adding laziness; and by reusing an existing internal method in a few places we were able to improve the inlining done by HotSpot. There were still a few methods, less critical ones, that were slightly slower as a result of the laziness changes, but even on general benchmarks and performance tests the changes produced a net win. So far, I haven't identified any case where the changes have produced a significant undesired effect.

Conclusion

When evaluating the memory usage of a JVM application, there is more to consider than just the maximum memory usage. Because the JVM uses garbage collection for memory management, you must also consider the memory allocation rate and the garbage collection pressure. *Memory allocation rate* refers to the rate at which the application allocates new objects and the size of those allocations. Applications vary widely in their allocation rate, and it is often an important factor in their throughput. Related to allocation rate is the amount of effort that must be spent to garbage-collect unused objects. *Garbage collection pressure* refers to how much throughput you must sacrifice for garbage collection to ensure that the application always has sufficient free memory to run. Generally speaking, most reductions in allocation rate also reduce the amount of garbage collection necessary.

In typical framework applications, the lazy initialization changes to ArrayList and HashMap produced modest 1 percent to 2 percent improvements in memory usage and allocation rate and barely measurable performance gains. Just as important, no applications had increased memory usage or reduced performance. In a smaller number of applications, I found up to a 20 percent reduction in memory usage and a similar reduction in memory churn. These dramatic benefits for some applications while simultaneously providing small benefits to most applications and no known negative impacts made the lazy initialization changes an important improvement.

Considering again the RestRequest example, how would the laziness changes to ArrayList and HashMap affect its behavior and performance? The RestRequest fields are still unconditionally initialized with final `ArrayList` and `HashMap` instances. This means that the usage of `RestRequest` is unchanged. No user programs have to add checks for any of its fields being null. In practice, though, we are likely to see that many of the collection instances created will be lazily empty—they will have deferred creation of their element arrays, providing savings of both memory and CPU cycles.

In any software system that has been refined over 20 years, such as Java, it's difficult to find any implementation change that is a unilateral benefit for all cases. In contemplating this change, I made sure that the normal uses of `ArrayList` and `HashMap` were not negatively affected. The analysis of performance on critical methods like `HashMap.get()` typically examines the cost of every Java bytecode and every CPU cycle—these are methods that are going to be run trillions of times per year across millions of JVMs. Some slight movement of performance cost, moving costs from one execution path to a different, later path, would be acceptable, but any degradation in performance would need to be inconsequential and would probably need to be offset by much larger performance gains elsewhere.

The analysis of the problem began by looking at application and framework behavior in the hopes that something could be done to reduce the cost of unused collections. This type of top-down performance analysis is, by far, the best approach to improving application performance.

Other opportunities for using laziness have been considered for the Java Collections Framework. The most desirable changes would be how `HashMap`s are built and resized. The typical usage pattern of `HashMap` suggests that the implementation would benefit from using different data structures for small maps (and for larger maps before the first `get()` operation).

There are other examples of laziness within the Java class library. The most common is to cache the result of hash code computations inside the `hashCode()` method. This is used with stellar performance benefits by `String` and other classes. Other caching cases have also been added. Some of these caches improve performance by avoiding repeated work; others save memory by reusing the same data structures for multiple operations. Additional caching cases and other lazy optimizations can be added if proved beneficial in future Java updates. There also have been times when a cache was wasteful or actually required too much effort to maintain and it was removed from the implementation. In most cases, because they don't involve API changes, laziness improvements to the Java libraries can be added with significant benefit and little impact.

Java 8 also introduced a significant new lazy implementation, the Streams API. This library utilizes laziness as a core principle and frequently delivers much better performance than simpler declarative approaches.

Laziness is an important optimization that has had substantial benefits in the Java libraries. You should strongly consider it if you need to improve the performance of a library that's in use by others and where many of the principal optimizations, such as algorithm refinement, have already been implemented. `</article>`

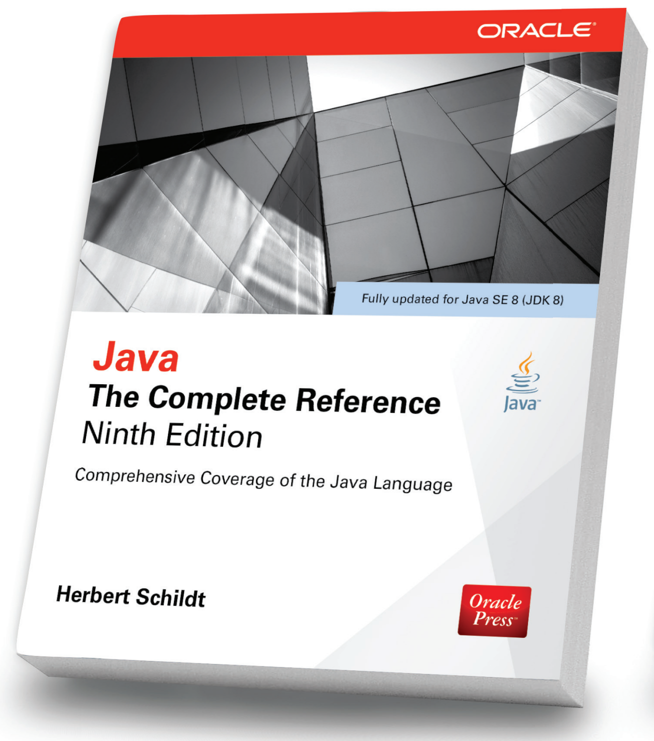
Mike Duigou (@mjduigou) works on Java-based ocean-going robots at Liquid Robotics. He was previously a developer on the Java Core Libraries team at Oracle and contributed to the core collections and Java 8 lambda libraries. Duigou has also enjoyed working on autonomous cars, dancing robots, and industrial real-time applications.

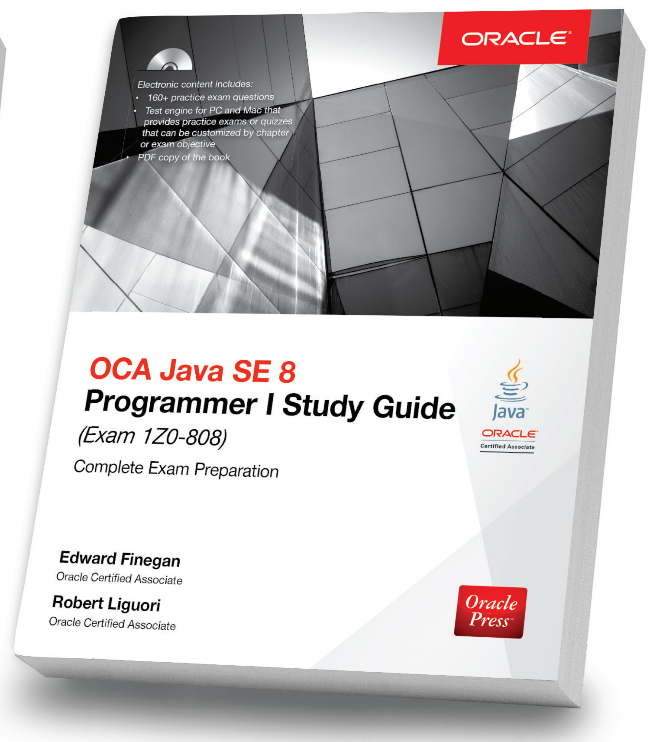
learn more

How laziness affects the size of an ArrayList allocation

ORACLE
Java
Raspberry Pi with Java: Programming the Internet of Things (IoT)
Embedded Application Development for Home and Industry
Stephen Chin with James L. Weaver
Oracle Press

ORACLE
Introducing JavaFX 8 Programming
A Fast-Paced Guide to JavaFX GUI Programming Fundamentals
Herbert Schildt
Oracle Press


ORACLE
Fully updated for Java SE 8 (JDK 8)
Java The Complete Reference Ninth Edition
Comprehensive Coverage of the Java Language
Herbert Schildt
Oracle Press


ORACLE
OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)
Complete Exam Preparation
Edward Finegan Oracle Certified Associate
Robert Liguori Oracle Certified Associate
Oracle Press

CÉDRIC BEUST

Annotations: An Inside Look

How annotations work, how best to use them, and how to write your own

Annotations appeared on the Java platform for Java 5 more than ten years ago, and they have become an integral part of the ecosystem. In this article, I go over a brief history of how and why annotations came about and then dive into technical details explaining how they operate, how they are best used, and how to write your own.

Origins

The idea of adding metadata to source code is quite old and comes from the simple realization that very often the code you write doesn't contain all the information a tool needs to do its job. On the Java platform, an early form of annotations began to appear before Java 5, but because the language did not officially support them, developers resorted to the unlikely Javadoc tool to add annotations to their code. Specifically, two tools brought annotations into the spotlight in the early 2000s:

- EJBGen, which enabled developers to add Javadoc annotations to their source code and which then generated the complicated EJB XML descriptors
- XDoclet, which took EJBGen's idea to the next level by providing a general framework for using Javadoc tags as annotations for any domain, not just for EJBs

These two tools became popular very quickly and opened the door for annotations to become officially supported by the JVM. JSR 250, Common Annotations for the Java Platform, was created specifically for this purpose and was scheduled to ship with Java 5. The idea was to make annotations type-safe and extensible so that developers could easily write their own.

It's interesting to note that since 2004, there has been only one major update to annotations in the JDK: JSR 308, which added more locations where annotations could be placed. But that's pretty much it. Today you are still using the same annotations as specified in 2005 with hardly any modifications. (JSR 308 added some minor utility that was discussed in the March/April 2014 issue.) It's hard to deny that JSR 250 has been a stable success that enabled many innovations, which I'll discuss shortly.

When to Use Annotations

Like all tools, annotations should be used judiciously. They are a great match for a certain category of problems but a poor choice when key conditions are not met. The main alternative to annotations is configuration files. Such files can hold the metadata that your code can't contain, just like annotations, so how do you decide whether specific metadata should be stored in annotations or in an external file?

The general rule of thumb is: If the metadata is tied to a Java element (method, field, variable, class, package, and so on), then it should be placed in an annotation. Otherwise, it should be stored in a configuration file.

As I mentioned, EJBs were the first target for annotations because their deployment descriptors were complicated XML files that referenced methods in the code and added extra information to them. This approach was error prone because refactoring code (such as renaming a method) might not be reflected in the deployment descriptor, and the application now fails to work. Instead, annotating the method that is the target of the metadata is much safer (starting with the fact that now you are no longer duplicating the name of that method—a violation of the DRY principle: Don't Repeat Yourself).

Here are some other good examples of annotation uses:

- **Code correctness.** Annotations such as `@Nullable`, `@Deprecated`, and `@Override` add important semantic information to methods and fields that the compiler can enforce.
- **Test methods.** Before TestNG and annotations came along, JUnit was using reflection to indicate that a method was a test method, which required specific naming conventions. With an annotation, it's no longer necessary to use naming conventions.
- **Persistence** (for example, Hibernate). You can annotate fields and methods in order to tie them to data stored in the database.
- **Dependency injection.** Classes that need to be injected can be annotated as such along with fields and parameters.
- **Graphical toolkits.** As an example, Android describes graphical layouts using XML; with annotations you can now directly tie graphical elements (text views, buttons, and so on) to the field that holds their reference.

Note that all these examples share the same characteristic: tying information to Java elements. In contrast, here are a few examples of metadata that are not good fits for Java annotations:

- Deployment information such as host names, ports, passwords, and other authentication details
- Connection pools informing your application how to connect to a database
- Parameters describing how an application should be launched or what kind of information is accessible at runtime

Important Annotations

Annotations are pretty much unavoidable in modern Java and plenty of libraries provide their own, but there are a few that stand out and that you should be using regularly.

@Nullable and @Nonnull (javax.annotation). These annotations can be placed on fields and method parameters, and they indicate whether these variables can be null. They are extremely useful, and a lot of tools on top of the Java compiler (javac) recognize them (starting with the major IDEs). You should use them at every opportunity. You will quickly notice the number of null pointer exceptions in your codebase sharply decreasing.

@Override. You are probably already familiar with this annotation because it has been mandatory since Java 6, and for good reason. This annotation must be placed on any method overriding a method from a parent interface or class. It prevents you from accidentally overriding a method or, conversely, from thinking you overrode such a method but did not because of a typo.

@FunctionalInterface. This is a new addition to Java 8. It makes sure that the interface so annotated is indeed a functional interface—that is, an interface with exactly one abstract method. The idea behind this annotation is that if one day you or someone on your team accidentally adds an abstract method to that interface, the compiler will issue an error.

@SuppressWarnings. This annotation is self-explanatory. Warnings are usually extremely useful, and you should never turn them off globally. However, it's occasionally useful to turn them off for specific statements or expressions when you know that your code is safe but the compiler doesn't.

Writing an Annotation

Let's take a look at a popular annotation: `@Test`. Here is its definition:

```
@Retention(RUNTIME)
@Target({METHOD})
public @interface Test {
```

Each line in the snippet above is important. Let's go over them one by one, starting from the bottom:

```
public @interface Test {
```

The syntax here is specific for annotations. Annotations look like Java classes but have several restrictions, which we will cover shortly.

```
@Target({METHOD})
```

This annotation specifies what Java elements can be annotated. `METHOD` is part of the `java.lang.annotation.ElementType` enum, which lets you define other locations for annotations such as `CLASS`, `CONSTRUCTOR`, and so on.

```
@Retention(RUNTIME)
```

This annotation indicates whether your annotation will be preserved in the class file or discarded by the compiler. If you want to be able to look up your annotation via reflection, the retention should be set to `RUNTIME`. The other options can be found in the `java.lang.annotation.RetentionPolicy` class.

Content of an Annotation
It is possible to pass additional parameters to annotations:

```
@Test(description = "Verify that bug #121 is fixed")
@Table(name = "ACCOUNTS")
```

These additional parameters are called *attributes*. They are defined as methods inside the declaration of your annotation:

```
public @interface Test {
    String description() default "";
}
```

Attributes are methods that don't have a body and that can optionally be assigned a default value. If you fail to use the `default` keyword, then that attribute needs to be specified when the annotation is used; otherwise, the compiler will issue an error. An important restriction on attributes is that they need to be constants: primitive types or a string (and they can't be null).

There are two interesting details that were included in the specification in order to reduce the amount of verbosity found in code using annotations.

If the annotation defines an attribute with the special name value, then you can specify that attribute without the word `value`. The following annotation:

```
public @interface Person {
    String value();
}
```

can be written as

```
@Person("John")
```

instead of the more common

```
@Person(value = "John")
```

In a similar vein, attributes of type `Array` can use a shorthand version when that array has only one element:

```
public @interface Languages {
    String[] value();
}
```

can use

```
@Languages("English")
```

instead of the more verbose

```
@Languages(value = { "English" })
```

These syntactic shortcuts were designed to reduce the boiler-plate code necessary when using annotations.

Annotations in Action

Now that you can define simple annotations, how do you actually use them? By design, annotations defined by third-party developers are completely ignored by the compiler. There are a few specific exceptions that the compiler acts upon (such as `@Deprecated`), but as a rule, annotations will never modify the semantics of the code to which they are applied. Therefore, the only way to make use of annotations is to write a tool that will act upon them.

There are two ways such tools can be written: as external tools or as annotation processors.

External Tools

External tools are the simplest approach to processing exceptions: you implement a separate application with its own `main()` method, and users of your annotations simply need to run this tool on their classes. This is the approach used by TestNG, JUnit, Guice, and other well-known tools. Such tools are typically run as part of your build, and the output of these tools can be quite varied: source files, documentation files, XML, and so on. There is really no limit on what these tools can do.

The JDK comes with an API to look up annotations in class files that is sprinkled throughout the reflection package. For example, you can either obtain all the annotations on a given class or only retrieve specific ones. Consider the following code:

```
@Languages({ "English", "French"})
class MyClass { … }
```

We can look up the annotation as follows:

```
Languages[] languages =
    getClass().getAnnotationsByType(Languages.class);
for (Languages language : languages) {
  System.out.println(
      "Languages spoken: " + language.value());
}
```

Annotation Processors

Annotation processors are a relatively recent addition to the JDK, and they have opened up a whole new level of innovation in the annotation field. Annotation processors were born from the observation that a large proportion of tools that process annotations generate Java source files, which then need to be compiled. Therefore, it appeared useful to integrate such processing inside the Java compiler itself so that the process could be streamlined.

The idea behind annotation processors is to declare them to the compiler so that it will invoke the processors first and then automatically compile the resulting output. Then the compiler resumes its usual process after adding your compiled classes to its classpath.

The API is a bit different from the reflection code I just covered, with a few variations. For example, instead of you looking up annotations, the compiler notifies you whenever it encounters an annotation. This approach is much more efficient.

Writing annotation processors is a bit more involved and would require a full article of its own. So for now, I'll explain the value of annotation processors.

Generating source code is not a new practice on the JVM. But because annotation processors hook directly into the compiler, a lot of the pain in building and processing the output is alleviated. There are several mechanisms that make running annotation processors completely seamless. For example, all it takes is for the JAR file containing your processor to be on the classpath for javac to detect it and run it.

Annotation processors have become particularly important on Android because they enable library developers to replace reflection calls (which are slow and therefore a special concern on Android) with direct calls. On top of that, the generated source code makes it possible to statically verify that your code is correct, something that can't be achieved with reflection.

Conclusion

Annotations are an integral part of the Java platform. They have allowed the language and the ecosystem to evolve in innovative and productive directions that wouldn't have been possible otherwise. It's important to understand how they work and to know what they are capable of so that you can make informed decisions about using them in your own codebase. </article>

Cédric Beust (@cbeust) has been writing Java code since 1996, and he has taken an active role in the development of the language and its libraries through the years. He holds a PhD in computer science from the University of Nice, France. He was a member of the Expert Group that designed annotations for the JVM.

learn more

Oracle's Java annotations tutorial

Checker Framework (which uses annotations heavily to check code)

Oracle Cloud Services for Java Developers

Oracle has been rolling out cloud services for developing and deploying Java applications. These services complement existing services that Oracle offers at all the major cloud tiers: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). With these rollouts, there are now multiple cloud services of interest to Java developers, of which these three will be discussed in technical detail in future issues.

Oracle Application Container Cloud provides rapid self-service provisioning of dedicated and isolated Java SE and Node runtime application containers in the cloud. These containers run Oracle JDK (version 7 or 8), which includes Oracle Java Flight Recorder, a tool that is not available in the standard JDK. The solution also offers Node.js, which is the server-side JavaScript environment. Billing is done either by the month or by the hour and billed per gigabyte of RAM.

Oracle Developer Cloud Service is a free entitlement of Oracle Java Cloud Service (see next item) and is described as a "PaaS environment for the enterprise." It includes instances of Git, Maven, Hudson (the continuous integration tool), a tasks tool, and a wiki. The Hudson instance allows three concurrent builds.

Oracle Java Cloud Service offers Oracle WebLogic Server (either 11*g* or 12*c*) running either in a cluster or on dedicated virtual machines. This service also offers Oracle Coherence caching and in-memory data grid as an option. An additional SaaS Extension enables integration with Oracle Software as a Service (including Oracle Sales Cloud, Oracle Service Cloud, and Oracle Marketing Cloud).

MICHAEL KÖLLING

PHOTOGRAPH BY JOHN BLYTHE

Making the Most of Enums

Anytime you have a set of known constant values, an enum is a type-safe representation that prevents common problems.

Enumerations—or *enums* for short—are Java constructs that are not used as much as they should be. They aren't one of those big, bold, buzzy concepts that get people excited or force themselves on you. Rather they quietly improve code, making it more reliable and more readable.

If you're new to Java, then it's entirely possible that you're writing good, functioning code but are not using enums. If so, you're not alone.

There are several reasons why some developers don't use them. First, early versions of Java did not have enums. Some programmers may have learned Java before Java 5, when enums were added to the language, and they never got around to changing their habits. Others may have come from different languages that did not support enums. And lastly, you might not have felt the need to use them because you were perfectly able to solve your problems without them. None of these are good reasons to continue ignoring them.

Enums enable you to make your code significantly better: more robust, more type safe, less error-prone, and more elegant. And these things matter. So sit back and read on.

When and Why to Use Enums

Let's examine the use of enums with an example. Suppose you want to write a text-based adventure game—something similar to Colossal Cave Adventure or Zork, two classic computer games. Then you will have a set of command words that the user can type in. And let's say the valid command words are go, look, take, help, and quit.

Somewhere in your code, you are likely to have a definition of those command words. In a straightforward first implementation, they might be defined in an array of strings, like this:

```
private static final String[] validCommands = {
    "go", "look", "take", "help", "quit"
};
```

Somewhere else in your program, you will have some code that reacts to these words being entered. The code then calls the right method to act on them. In this code snippet, I assume that the `String` variable `commandWord` holds the word that was typed in.

```
switch (commandWord) {
    case "go":
        goRoom(secondWord);
        break;
    case "look":
        look();
        break;
    case "take":
        takeItem(secondWord);
        break;
    case "Help":
        printHelp();
        break;
    case "quit":
        quit();
        break;
}
```

(An alternative would be to define a sequence of int constants for these commands, and then translate the input string to a number and switch on the int constant. This popular variant has the same problems that I discuss with our solution here.)

What's wrong with this solution? There are really two separate fundamental problems that immediately stand out: type safety and internationalization.

Let's deal with type safety first. The short code segment presented above is quite straightforward and easy to understand. It works, right? In fact, it doesn't. There is a bug in the code. Did you spot it?

The problem is that the help command has been mistyped in the switch statement as Help. Even though it was our intention that the commandWord should only ever be one of the strings listed as valid commands, there is nothing stopping us from assigning invalid commands or comparing it to invalid commands. Because the declared type is String, any string will do. In effect, our type system is not good enough. The declared type does not properly describe the set of acceptable values, and (logically) illegal values can be used without the type system being able to detect this.

Enums to the Rescue

To avoid this problem, we can rewrite our code using enums. We first write an enum declaration:

```
public enum CommandWord
{
    GO, LOOK, TAKE, HELP, QUIT
}
```

This declaration should be treated like a class and written in its own file. It defines the type CommandWord and the five listed names as valid values for that type. In other classes, we can then declare variables of this type and assign values. For example:

```
CommandWord command = CommandWord.GO;
```

And importantly, we can rewrite our switch statement to the following:

```
switch (commandWord) {
    case GO:
        goRoom(secondWord);
        break;
    case LOOK:
        look();
        break;
    case TAKE:
        takeItem(secondWord);
        break;
    case HELP:
        printHelp();
        break;
    case QUIT:
        quit();
        break;
}
```

The definition of the command words in this version (as an enum, instead of a string array) is not only clearer and simpler, it also creates type safety: if you now mistype a case label or a value in an assignment, the compiler will detect this and notify you. This is a real win—we have our strong type system back that Java was designed for.

By the way, we can also use the double equals symbol (==) for checking equality, instead of the .equals() method that we had to use with strings:

```
if (command == CommandWord.QUIT) ...
```

What Really Is an Enum?

Some people use enums only as described here and think of them as similar to int constants: named values that can be assigned and recognized later. But this is not the complete truth, and if you stop here, you have only scratched the surface and are missing out on some of the best features.

Enum declarations are full classes, and the values listed are constant names referring to separate instances of these classes. The enum declaration can contain fields, constructors, and methods, just like other classes. Here is an extended version of the previous enums:

```
public enum CommandWord
{
    GO("go"), LOOK("look"), TAKE("take"),
    HELP("help"), QUIT("quit");
    private String commandString;

    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    public String toString ()
    {
        return commandString;
    }
}
```

The important aspects are the following:

- Enum declarations are classes, and enum values refer to objects.
- For every declared enum value, an instance of the class is created and assigned to that value.
- No other instances of this class can be created later.
- Every different enum value will refer to a different object, and the same value will always refer to the same object; this cannot be changed.
- Enums create their own namespace, so different enum classes may use the same value, but these are kept separate. If, for example, I have an enum class BoardGames, the enum values BoardGames.GO and CommandWords.GO are separate and do not interfere with each other.

The last aspect—that no other instances may be created—is ensured by making the constructor private. It is not necessary to declare this explicitly: the constructor is automatically private, and it is an error to try to make it public.

The previous code will generate five enum objects—one for each value. And any reference in other code to CommandWord objects can be to only one or more of these enums. Any attempt to create other objects will generate a compile-time error.

Enums may contain any number of fields, constructors, and methods. The fundamental difference when compared with "normal" classes is in how enum instances come into existence. While other classes start without any instances and provide a constructor for clients to create as many objects as they like, enums provide no constructor (to the outside), and instead provide a set of ready-made instances.

The fact that enum values are objects, not ints, is important. It means that enums provide not only identity but also state and behavior.

The Full Truth

The first question that now comes to mind is this: If the constructor cannot be called from the outside, what is it used for?

The answer lies in the modified syntax we have used for enumerating our enum values. Instead of just

```
GO
```

as in our first version we have now written

```
GO("go")
```

This extension—effectively adding a parameter list to the enum value—invokes the enum's constructor. The expression within the parentheses is the actual parameter passed to the constructor. The enum object is still of class CommandWord, as before, but we are now storing a string attribute inside it. And the value for this attribute is passed in to our enum object via its constructor. We can store any number and type of attributes inside an enum object—that is, in instance fields—just like in any other object.

Read our CommandWord definition again—it should all slowly come together and start to make sense now.

The great advantage of this scheme is that we can now use a String again to recognize the typed word (for example, "help") by comparing the input string against the command strings stored inside our enums, but our program logic is independent of these strings.

Earlier, I mentioned that another problem with our first version was *internationalization*: If we decide to translate our program into a different language (let's say the "help" command is now "hilfe") we run the danger of introducing errors. If we just change the command words in the array, the program will compile but not function; none of the commands will be recognized, but we don't get an error. The problem is that the strings are not only used for input but also for the program logic. That is bad.

In our new enum version, that problem has been resolved. The actual command strings are mentioned only once; if they change, they need to be changed only in one location, and the program logic works with logical values—the enum constants—that will continue to work. (In practice, the input commands would be read out of a locale-dependent text file, but the principle is the same.)

Under the Hood

Enums are really implemented as classes, and enum values are their instances. There is little special about this, and knowing this helps us understand how they work and what we can do with them.

Enum classes all automatically inherit the Java standard class Enum from which they inherit some potentially useful methods (it also means that they cannot extend another class).

The inherited methods you should know about are name(), ordinal(), and the static method values().

The name() method returns the name exactly as defined in the enum value. The ordinal() method returns a numeric value that reflects the order in which the enums were declared, starting with zero. For example,

```
CommandWord cmd = CommandWord.GO;
System.out.println(cmd.name());
System.out.println(cmd.ordinal());
```

will print

```
GO
0
```

In practice, these two methods are much less useful than you might first think. Your code typically should not depend on the actual enum name (so the name() method is not often useful; it is much better to override and use the toString method for that purpose), and if you write your code well you will rarely need the ordinal number.

The static values() method is more often useful. It returns an array of all enum values and can be used to iterate over them. Here's an example.

```
CommandWord[] ca = CommandWord.values();

for (CommandWord cw : ca) {
    System.out.println(cw);
}
```

Or, if you are familiar with Java 8's streams, you can also write the following:

```
Arrays.stream(ca).forEach(System.out::println);
```

The Enum Singleton Pattern

Once you understand how enums are really implemented under the hood (most importantly, that they are just classes with a different instance creation mechanism), you might discover some helpful ways to use them. One example I use regularly is to employ an enum to implement a singleton pattern.

A singleton is employed to ensure that only a single instance exists of a given class. It is often written by creating the instance in the class, making the constructor private, and providing a static factory method to hand out the instance, as in the following example:

```
public class Singleton {
    private static Singleton instance =
        new Singleton();

    public static Singleton getInstance()
    {
        return instance;
    }

    private Singleton()
    {
        ...
    }
}
```

The singleton instance can then be accessed from the outside by writing

```
Singleton s = Singleton.getInstance();
```

A nice alternative is to use an enum to define the singleton:

```
public enum EasySingleton {
    INSTANCE;
}
```

No more work is needed, and the instance can easily be accessed from client code:

```
EasySingleton s = EasySingleton.INSTANCE;
```

Fields and methods can still be added to the singleton class as before. Enum instance creation is by default thread-safe, so this method is safe to use in a multithreaded application.

Conclusion

I hope this short introduction has demonstrated the advantages of enums. Anytime you find yourself defining a set of constant values, you should think of enums as your preferred way of representing them. This choice gives you type safety, support for internalization, and warnings at compile time about possible coding errors. Overall, it will make your code more readable and less prone to errors. </article>

Michael Kölling is a professor at the University of Kent, UK. He has published two Java textbooks and numerous papers on object orientation and computing education topics, and is the lead developer of BlueJ and Greenfoot, two educational programming environments. Kölling is an Oracle Java Champion, a UK National Teaching Fellow, a Fellow of the UK Higher Education Academy, and a Distinguished Educator of the ACM.

learn more

Oracle Java tutorial on enums

JOSH JUNEAU

What's New in JPA: The Criteria API

Create queries and update databases with Java entity classes and fields, rather than with strings of SQL.

Java EE applications usually rely on a data repository for storage of application data. An application data repository can be a typical RDBMS (relational database management system) such as Oracle Database, MySQL, PostgreSQL, or Microsoft SQL Server. Often in modern applications, though, NoSQL databases such as MongoDB, Cassandra, Couchbase, Oracle NoSQL Database, or Neo4j are used as back-end data repositories. Regardless of the data repository solution for a Java EE application, the Java Persistence API (JPA) plays a vital role for the storage and retrieval of application data.

In my article in the May/June 2015 issue of this magazine, "What's New in JPA," I took a look at a handful of the features that were added to JPA 2.1, which is part of the Java EE 7 release. In that article, I covered features such as attribute conversion, schema generation, named stored procedures, and more. I also touched briefly on a new feature for the Criteria API: bulk operations. In this article, I do a deeper dive into the Criteria API to explain more of its benefits, which include the ability to create queries without using strings of text. I also take a deeper look at some of the newer features that were added as part of JPA 2.1.

Constructing JPA Queries

As you may know, there are a handful of ways to query, update, and delete data using JPA. Let's take a brief look at each of them to refresh our memory, and then we'll explore more detail about the Criteria API. In these examples, I'll use the same basic `SELECT` query for each example so that you can gain a better understanding of each. Namely, I'll select all records from the `POOLS` database table. **Listing 1** demonstrates each of the techniques for performing this simple query in JPA.

Listing 1.

```
// Named Query
public List<Pool> getAllPoolsNamed(){
    return em.createNamedQuery("Pool.findAll")
        .getResultList();
}

// Native Query
public List<Pool> getAllPoolsNative(){
    List<Pool> pools = (List<Pool>)
        em.createNativeQuery(
            "select * from Pool",
            com.acme.acmepools.entity.Pool.class)
              .getResultList();
    return pools;
}

// JPQL Query
public List<Pool> getAllPoolsJPQL(){
    return em.createQuery("select o from Pool o")
             .getResultList();
}
// Criteria API Query
public List<Pool> getAllPoolsCriteria(){
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery cq = cb.createQuery();
    Root<Pool> from = cq.from(Pool.class);
    TypedQuery<Pool> typedQuery =
        em.createQuery(cq.select(from));
    return typedQuery.getResultList();
}
```

Named queries, which can be defined in a few different ways, basically map a typed query to a specified name. The query can later be called upon by name, and JPA will then perform the assigned query. *Native queries* are strings of text that formulate a SQL query using the native syntax of the target database platform. Native queries have a tendency to be less portable. One of the most frequently used techniques for querying with JPA is to use the Java Persistence Query Language (JPQL). Similar to native queries, *JPQL queries* are constructed of strings of text to formulate a query using JPQL syntax. JPQL is based on the abstract schema of entity classes that have been registered with a persistence context. All related objects of those entity classes can also be managed via JPQL. The advantages of using JPQL are that the syntax is similar to standard SQL, and JPQL queries are portable regardless of the underlying datastore—even if it's NoSQL.

Lastly, the Criteria API can be used to construct queries in a strongly typed manner using the Java entity classes and fields, rather than using strings of text. Similar to JPQL, the Criteria API is also based upon the abstract schema of entity classes, so it works in concert with the persistence context. Looking at the end of Listing 1, it is plain to see that the Criteria API can be rather verbose. However, there are some significant advantages to using the API. Let's take a look at those.

An advantage is that Criteria API queries are completely portable, meaning that the queries are independent of the underlying datastore.

Getting Started with the Criteria API

The Criteria API allows you to build database queries in a strongly typed manner from objects, using all Java code. Doing so cuts down on the possibility for errors, because you no longer need to worry about making sure that typed JPQL or SQL queries are error-free. Another advantage is that these queries are completely portable, meaning that the queries are independent of the underlying datastore. The API also contains an extensive Metamodel API, which assists in producing type-safe queries.

The Criteria API query shown at the bottom of Listing 1 follows the procedural set of steps that are typically followed to create a query and retrieve data. Let's walk through each step in detail.

First, create a `CriteriaBuilder` object from the `EntityManager` or `EntityManagerFactory`:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

Next, create a `CriteriaQuery` object from the `CriteriaBuilder`:

```
CriteriaQuery cq = cb.createQuery();
```

The `CriteriaQuery` can be used to designate against which entities the query will be executed. To do this, call its `from` method and pass the class object for the entity; it will return a `Root` object of the given entity class type. The `Root` represents the entity from which all navigation will originate. This example navigates over the `Pool` entity:

```
Root<Pool> from = cq.from(Pool.class);
```

Then use the Root to create a TypedQuery object. To do that, call the EntityManager.createQuery() method, and pass the CriteriaQuery.select method, along with the Root object:

```
TypedQuery<Pool> typedQuery =
    em.createQuery(cq.select(from));
```

Lastly, retrieve the results by calling the TypedQuery .getResultList() method:

```
typedQuery.getResultList();
```

Once the query has been executed and results are returned, any fields from the specified entity can be navigated, because they are all available in the object. Similarly, any related records are made available. For instance, Listing 2 shows how to use a Pool. The Pool entity contains a Collection<Customer> field that can be used for retrieving Customer objects that own a Pool of the specified type. In the entity class, the fetch mode is set to FetchType.LAZY, meaning that the Collection<Customer> is available only when the getCustomers() method is explicitly called upon. If I wanted to have the Collection retrieved at the same time as the Pool objects, the fetch mode should be set to FetchType.EAGER.

Listing 2.

```
@Entity
@Table(name = "POOL")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Pool.findAll",
                query = "SELECT p FROM Pool p"),
    . . .
    })

. . .
public class Pool implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "ID")
    private Integer id;
    @Size(max = 10)
    @Column(name = "STYLE")
    private String style;
    @Size(max = 10)
    @Column(name = "SHAPE")
    private String shape;
    // @Max(value=?)  @Min(value=?)//if you know
    // the range of your decimal fields, consider
    // using these annotations to enforce
    // field validation
    @Column(name = "LENGTH")
    private Double length;
    @Column(name = "WIDTH")
    private Double width;
    @Column(name = "RADIUS")
    private Double radius;
    @Column(name = "GALLONS")
    private Double gallons;
    @Column(name = "SHALLOW_DEPTH")
    private Double shallowDepth;
    @Column(name = "DEEP_DEPTH")
    private Double deepDepth;
    @OneToMany(mappedBy = "pool",
               fetch=FetchType.LAZY)
    private Collection<Customer> customer;

    public Pool() {
    }

    . . .
}
```

Working the Metamodel

Now that you know how to create a standard query and retrieve results, it is time to learn how to really exploit the power of the Criteria API. To dig into the API and maintain type-safe operation, work with the metamodel of the application entities. All entity classes that are used with the Criteria API have a class that is either constructed at runtime behind the scenes or statically developed via the use of annotations. A metamodel allows direct access to each of the fields within the entity without passing the string-based name of the field. This approach creates a type-safe programming model. **Listing 3** shows the statically typed metamodel class for the `Pool` entity.

Listing 3.

```
import javax.persistence.metamodel.*;

@StaticMetamodel(
     com.acme.acmepools.entity.Pool.class)
public class Pool_ {

    public static volatile
        SingularAttribute<Pool, Integer> id;
    public static volatile
        SingularAttribute<Pool, String> style;
    public static volatile
        SingularAttribute<Pool, String> shape;
    public static volatile
        SingularAttribute<Pool, Double> length;
    public static volatile
        SingularAttribute<Pool, Double> width;
    public static volatile
        SingularAttribute<Pool, Double> radius;
    public static volatile
        SingularAttribute<Pool, Double> gallons;
    public static volatile
        SingularAttribute<Pool, Double> shallowDepth;
    public static volatile
        SingularAttribute<Pool, Double> deepDepth;
    public static volatile
        CollectionAttribute<Pool, Customer> customer;
}
```

There are two types of metamodel classes: canonical and noncanonical. The class in **Listing 3** shows a noncanonical metamodel class, which is explicitly created by the application developer. However, development of such a noncanonical metamodel class might compromise portability across JPA providers. Each provider, in turn, is expected to generate a canonical metamodel class for each entity. Therefore, unless there is a good reason to develop the metamodel class, it might be best to use the canonical class that is generated at compilation time.

There are several ways to use the metamodel to achieve the desired result. The `CriteriaBuilder` can be used to obtain a number of `Predicate` or `Expression` objects that are used to filter results. See the JavaDoc for more details. A `Predicate` is used to set a `boolean` condition, and an `Expression` is used to build the condition. In the case of our `Pool` entity, suppose that we wanted to set a condition to retrieve all entities where the shape was rectangular. In this case, to set the condition, use the `CriteriaBuilder` to obtain a `Predicate` object, passing the expression that sets the condition to get the `Pool_.shape` field, along with the value of the condition (`"RECTANGLE"`) as the second argument. The `Pool_.shape` reference is also known as a *path expression*.

```
Predicate condition =
    cb.equal(from.get(Pool_.shape), "RECTANGLE");
```

Note that if you were not using the Metamodel API, it would be possible to produce the same `Predicate` by passing the string-based name of the field within the condition, as shown below. However, this would increase the potential for runtime errors because the field name would not be checked during compilation.

```
// Using a string-based field name
// Not the preferred approach
Predicate condition =
    cb.equal(from.get("shape"), "RECTANGLE");
```

After obtaining the `Predicate`, the `CriteriaQuery` can then be altered by initiating a call to the `where()` method, passing the `Predicate`. The `where()` method acts as a modifier to the initial `CriteriaQuery` object, allowing the query to be altered to set one or more conditions. The `CriteriaQuery` also contains a number of other modifiers. See the JavaDoc for more details.

```
cq.where(condition);
```

Lastly, obtain the `TypedQuery` object by calling upon the `EntityManager`'s `createQuery()` method and passing the `CriteriaQuery` object. Finally, obtain the `ResultList` from the `TypedQuery`. **Listing 4** shows the complete example code.

Listing 4.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery();
Root<Pool> from = cq.from(Pool.class);
Predicate condition =
    cb.equal(from.get(Pool_.shape), "RECTANGLE");
cq.where(condition);
TypedQuery<Pool> typedQuery = em.createQuery(cq);
return typedQuery.getResultList();
```

In the case where there is more than one condition that needs to be applied to your query, simply create a new `Predicate` and then pass to the `CriteriaQuery.where()` method each of the `Predicate` objects separated by commas. **Listing 5** demonstrates a query that returns all `ROUND` pools that can hold more than 25,000 gallons of water. This particular listing certainly shows the benefits of the strongly typed Criteria API.

Listing 5.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery();
Root<Pool> from = cq.from(Pool.class);
Predicate condition1 =
    cb.equal(from.get(Pool_.shape), "ROUND");
Predicate condition2 =
    cb.gt(from.get(Pool_.gallons), 25000);
cq.where(condition1, condition2);
TypedQuery<Pool> typedQuery = em.createQuery(cq);
return typedQuery.getResultList();
```

For instance, if you were to pass a String to the `CriteriaBuilder.gt()` method, the compilation will fail because it expects a numeric value. It is easy to reconstruct this `CriteriaQuery` to make it more closely resemble SQL or JPQL syntax, if you desire. Rather than passing the `Predicate` conditions and calling upon the `EntityManager.createQuery()` separately, we would perform these same tasks using a builder pattern to produce a `TypedQuery` object. Moreover, we can create a `List` of `Predicates` or conditions to make the code more manageable. There are other methods that can be invoked within the query builder chain to perform ordering, capture distinct fields, and so forth. **Listing 6** demonstrates the same query that we used in **Listing 5**, but with this streamlined syntax, and sorted by the number of gallons.

Listing 6.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery();
Root<Pool> from = cq.from(Pool.class);
List<Predicate> conditions = new ArrayList();
conditions.add(
    cb.equal(from.get(Pool_.shape), "ROUND"));
conditions.add(
    cb.gt(from.get(Pool_.gallons), 25000));
TypedQuery<Pool> typedQuery = em.createQuery(cq
      .select(from)
      .where(conditions.toArray(new Predicate[] {}))
      .orderBy(cb.asc(from.get(Pool_.gallons)))
);
return typedQuery.getResultList();
```

Performing Joins

Often, applications require queries that return data from more than one database table or entity class. In the database world, we would use SQL joins to relate records from more than one table to each other. For instance, if we wanted to retrieve all customers of the Acme Pool company that had an `INGROUND` pool, we would join the `POOL` table with the `CUSTOMER` table in the `POOL_ID` column, because each `CUSTOMER` record contains a `POOL_ID` that relates to a model of pool that is stored in the `POOL` table. Such a SQL statement might look as follows:

```
select c.name
from customer c,
    pool p
where c.pool_id = p.pool_id
and p.style = 'INGROUND';
```

This query would return each customer's name, where the customer is on record as owning an `INGROUND` style pool. To perform a similar join using the Criteria API, simply retrieve a `Root` for the entity on which you would like to join, and then invoke the `Root.join()` method, passing the path expression for the field on which you would like to perform the join. This will return a `Join` object that you can then use within a selection.

```
Root<Pool> pool = cq.from(Pool.class);
Join<Pool, Customer> poolCustomers =
 pool.join(Pool_.customer);
```

In **Listing 7**, you can see the complete code, where the same join that was performed earlier using SQL is done using the Criteria API.

Listing 7.

```
public List<Customer> ingroundPoolCustomers() {
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Customer> cq =
        cb.createQuery(Customer.class);

    Root<Pool> pool = cq.from(Pool.class);
    Join<Pool, Customer> poolCustomers =
        pool.join(Pool_.customer);

    TypedQuery<Customer> query = em.createQuery(
        cq.select(poolCustomers)
          .where(cb.equal(pool.get(Pool_.style),
                          "INGROUND"))
    );
    return query.getResultList();

}
```

The Criteria API's Latest Features

In my earlier article on the new features in JPA, I briefly covered some features that were added to the Criteria API with the release of JPA 2.1 (in Java EE 7): bulk updates and bulk deletions. Now, I want to cover these new features in more detail. In JPA 2.1, the `CriteriaUpdate` and `CriteriaDelete` objects were added to the API, and the `CriteriaBuilder` was extended so that it could be used to produce these objects for performing bulk update and delete operations. Let's examine these.

Bulk updates. In some instances, it makes sense to apply an update to a large number of database records. Perhaps you need to update the cost for all customers in a particular city due to material increases, or maybe all customers with a specific pool style need to be updated to enable or disable maintenance due status. In the following example, I use the latter scenario so that all customers of a specified pool type will have maintenance either enabled or disabled when the update is executed.

To perform a bulk update, use a `CriteriaBuilder` to generate a `CriteriaUpdate` object that is set to the type of entity you want to update. In the following example, I want to update the `Customer` entity, so I call the `CriteriaBuilder.createCriteriaUpdate()` method, passing it the `Customer.class`:

```
CriteriaUpdate<Customer> customerUpdate
    = builder.createCriteriaUpdate(
        Customer.class);
```

The `CriteriaUpdate` object can then be used to indicate which field(s) of the entity will be updated, and under which conditions the update will occur.

In the following example, I am using a `Join<Customer, Pool>` object identified as `poolCustomers` to retrieve the style of pool that each customer owns. Therefore, I call the `CriteriaUpdate set()` method, passing the path expression to the fields I want to update, along with the values to set. Next, I call the `where()` method, which specifies the conditions under which the update is applied. Once again, I can use the `builder` pattern:

```
customerUpdate.set(
      customer.get(Customer_.currentMaintenance),
      enabled)
    .where(
      builder.equal(
        poolCustomers.get(Pool_.style),
        poolStyle);
```

Once all of the appropriate fields have been set and conditions have been put into place, I create a `Query` object by calling the `EntityManager createQuery()` method and passing the `customerUpdate`:

```
Query q = em.createQuery(customerUpdate);
```

Lastly, I invoke the `executeUpdate()` method to execute the update. The complete listing for this bulk update example is shown in Listing 8.

Listing 8.

```
public void updateMaintenanceByPoolType(
    String poolStyle, boolean enabled) {
      CriteriaBuilder builder =
          em.getCriteriaBuilder();
      CriteriaUpdate<Customer> customerUpdate =
          builder.createCriteriaUpdate(
              Customer.class);
      Root<Customer> customer =
          customerUpdate.from(Customer.class);

      Join<Customer, Pool> poolCustomers =
          customer.join(Customer_.pool);

      customerUpdate.set(
        customer.get(
          Customer_.currentMaintenance), enabled)
        .where(builder.equal(
          poolCustomers.get(Pool_.style),
          poolStyle));
      Query q = em.createQuery(customerUpdate);
      q.executeUpdate();
      em.flush();
}
```

Bulk deletions. There might be a circumstance in which it is appropriate to delete more than one record from a database in a single invocation. A bulk update can be performed in such cases, eliminating the need to perform multiple separate deletions using a looping mechanism, as was done in the past. For instance, suppose the Acme Pools company wanted to delete all customers whose pool size is greater than a set number of gallons. This would be a perfect case for using the bulk deletion feature.

In much the same way that bulk updates are implemented, a bulk deletion can be performed via a `CriteriaDelete` object. In this case, we would like to create a `CriteriaDelete` object of the `Customer` type, as follows:

```
CriteriaDelete<Customer> customerDelete
    = builder.createCriteriaDelete(
        Customer.class);
```

Because there are no values to set for a deletion, `CriteriaDelete` is simpler to use than `CriteriaUpdate`. To perform the deletion, call the `where()` method, passing the conditions that must be met for the deletion to occur. In this case, the number of gallons in the customer's pool must be more than the threshold. So, the `CriteriaBuilder gt()` method is called, passing the path expression to the `Pool_.gallons` field, along with the gallons threshold as the second argument:

```
customerDelete
    .where(builder.gt(poolCustomers.get(
                      Pool_.gallons),
           gallons));
```

Once again, simply create a query from the `CriteriaDelete` object, and then invoke the `executeUpdate()` method to perform the deletion. Listing 9 shows the code for this example.

Listing 9.

```
public void
  removeCustomerByGallons(double gallons) {
    CriteriaBuilder builder =
        em.getCriteriaBuilder();
    CriteriaDelete<Customer> customerDelete =
        builder.createCriteriaDelete(Customer.class);
    Root<Customer> customer =
        customerDelete.from(Customer.class);

    Join<Customer, Pool> poolCustomers =
        customer.join(Customer_.pool);

    customerDelete
        .where(builder.gt(
         poolCustomers.get(Pool_.gallons),
         gallons));
    Query q = em.createQuery(customerDelete);
    q.executeUpdate();
    em.flush();
}
```

Diving Deeper into the Criteria API

In the world of databases and SQL, there are several other operations and query options that can be used to produce the desired result. One example is the use of aggregate functions for performing calculations on data, and another example is fetching only distinct values. Moreover, SQL queries can include subqueries to be used for filtering results. All of these options are also available in the Criteria API.

Most of the aggregate functions can be performed by calling the `CriteriaBuilder` object, whereas ordering and grouping functionality can be done with a `CriteriaQuery` object. Such operations can be addressed using the same builder pattern that I've used throughout the examples in this article.

Because subqueries use a somewhat different syntax, let's take a look at a quick example. Suppose that we want to query all Acme Pools customers that contain a discount code, where the discount rate is higher than a specified amount. A standard SQL query for this situation might look as follows:

```
select *
from customer
where discount_code in
  (select discount_code
   from discount_code where rate > 7.00);
```

Now let's use the Criteria API to construct this same subquery. The `Subquery` object comes into play when working with subqueries. Much of the query is written in the same manner as others we've seen, but in this case, two separate queries are written and then combined to produce the result. Listing 10 demonstrates how to perform a subquery.

Listing 10.

```
CriteriaBuilder criteriaBuilder =
    em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery =
    criteriaBuilder.createQuery(Customer.class);
Root<Customer> from =
    criteriaQuery.from(Customer.class);

CriteriaQuery<Customer> select =
    criteriaQuery.select(from);

Subquery<DiscountCode> subquery =
    criteriaQuery.subquery(DiscountCode.class);
Root fromDiscountCode =
    subquery.from(DiscountCode.class);

subquery.select(fromDiscountCode.get(
                DiscountCode_.discountCode))
    .where(criteriaBuilder.gt(
     fromDiscountCode.get(DiscountCode_.rate),
     7000));
select.where(criteriaBuilder.in(
     from.get(Customer_.discountCode))
     .value(subquery));

TypedQuery<Customer> typedQuery =
     em.createQuery(select);
return typedQuery.getResultList();
```

Conclusion

The Java Persistence API is the foundation for performing database operations within a standard Java EE 7 application. Although there are a variety of ways to work with data, the Criteria API is the only one that allows the construction of type-safe queries, eliminating many of the runtime errors that occur from incorrect String queries. The API can be used to build complex queries and perform bulk operations, the latter of which are new features in the latest release of JPA. </article>

Josh Juneau works as an application developer, system analyst, and database administrator. He authored *Java EE 7 Recipes*, *Introducing Java EE 7*, and *JavaServer Faces: Introduction by Example* (all from Apress). He also produced a video course entitled *Mastering PrimeFaces* (Packt Publishing, 2015).

learn more

[The complete source code on GitHub for the AcmePools application]

[Java EE 7 tutorial]

[CriteriaQuery documentation]

[CriteriaBuilder documentation]

JULIEN PONGE

PHOTOGRAPH BY
MATT BOSTOCK/GETTY IMAGES

Golo

A fast, low-ceremony, easy-to-learn language for the JVM

I started writing what would become the Golo programming language in the summer of 2012. I was working on dynamic software modifications with my research colleagues, and we had proposed a JVM agent called JooFlux to inject changes and aspects into Java programs on the fly. It was based on JSR 292 (support for dynamic languages on the JVM) and the `invokedynamic` bytecode instruction. I ended up experimenting a lot with this bytecode trying to facilitate the design of dynamically typed languages on top of the JVM.

As I studied existing JVM languages that lacked `invokedynamic`, I thought it would be a good idea to create a language that has simple and easy-to-understand compiler and runtime codebases to enable experimenting with language advances, such as new bytecodes. Fast-forward a few years and Golo is now an incubating Eclipse Technology project where hobbyists who have no prior language development experience contribute to its development, and research derivatives have been based on it.

This article provides a tour of some of the features of the Golo programming language. It does not cover all the features, yet it should give you a good start.

A Verbose Start

Golo is a dynamically typed language. It supports imperative and functional idioms, and it integrates nicely with Java. To illustrate that, let me start with a simple program that creates a `java.util.ArrayList` from the Java standard APIs, and then iterates over the elements to print them. The constructions are deliberately close to what you would write in Java, but I will soon show how to make them more concise.

```
module javamag.Hello

import java.util

# A comment
function main = |args| {
  let elements = ArrayList()
  elements: add("Hello")
  elements: add("world")
  elements: add("!")
  let size = elements: size()
  for(var i = 0, i < size, i = i + 1) {
    print(elements: get(i))
    if (i < size - 1) {
      print(" ")
    }
  }
  println("")
}
```

The code above demonstrates a few things:

- The code is a *module*, which is the compilation unit in Golo.
- `import` statements help resolve symbols.
- Function parameters are passed between pipe symbols (`|`).
- There is no `new` operator for creating instances of Java classes; instead, the constructors are called as functions.
- `let` defines constant references, and `var` defines variable references.
- Instance methods are called using the `:` operator, as in

elements: size(), which calls the size method on the elements object. (The space after the colon is idiomatic, not mandatory.)
- Comment lines start with a # symbol.
- A Golo module can provide a main function as an entry point, which takes exactly one argument that is expected to be an array of command-line arguments, just like the main method in Java.

It is important to note that import statements are purely symbolic and are not checked at compile time. Had import java been used instead of import java.util, the code would have called the ArrayList constructor as util.ArrayList().

Golo provides a unique golo command-line tool. It also provides several subcommands, including compile, run, and golo. Compiling and then running the above program is as simple as the following:

```
$ golo compile hello-1.golo
$ ls javamag/
Hello.class
$ golo run --classpath . --module javamag.Hello
Hello world !
$
```

The golo subcommand compiles source code in memory, rather than to files, and then executes a module, which is by default the last .golo file:

```
$ golo golo --files hello.golo
Hello world !
$
```

Reducing Verbosity

Golo provides collection literals for creating lists, arrays, vectors, tuples, maps, sets, and ranges:

```
[1, 2, 3]
list[1, 2, 3]
map[[1, "a"], [2, "b"]]
[1..10_000]
# ...
```

These options allow me to revisit the previous example and introduce a foreach loop:

```
let elements = vector["Hello", "world", "!"]
foreach e in elements {
  print(e + " ")
}
println("")
```

Note that the foreach loop supports *conditional guards* with a when clause. The following example uses a range (the double dots) and prints only the odd numbers:

```
let ints = [1..100]
foreach i in ints {
  if i % 2 == 0 {
    println(i)
  }
}
```

The previous example can be rewritten using when:

```
let ints = [1..100]
foreach i in ints when i % 2 == 0 {
  println(i)
}
```

Golo provides *collection comprehensions* for all collection literals. This feature is reminiscent of the Python programming language, and here is a simple example:

```
let odds = [i foreach i in [1..100] when i % 2 == 0]
println(odds)
```

The previous code defines a tuple of the odd integer numbers between 1 and 100. As you can see, embedding a foreach clause that may also contain a when guard does this.

There can also be several foreach clauses, as the next example shows. It generates a collection of pairs of integers, expressed as a tuple of tuples:

```
let pairs = [ [i * 2, j * 3]
  foreach i in odds when (i >= 30) and (i <= 50)
  foreach j in [10..20] when j % 2 == 1 ]
println(pairs)
```

Golo also supports *destructuring* of its data types and collections:

```
let l = list["1", "2", "3", "4"]
let a, b = l
let head, second, tail... = l

let m = map[["a", 123], ["b", 456]]
foreach key, value in m: entrySet() {
  # ...
}
```

In the two examples above, a would be value "1", and b would be value "2". Similarly head, second, and tail would be, respectively, "1", "2", and list["3", "4"]. Destructuring is also useful when dealing with map entries and decomposes them as pairs of keys and values (a Golo map[...] literal yields a java.util.HashMap).

Higher-Order Functions

Like most recent programming languages, Golo supports higher-order functions. This means that functions can accept functions as parameters and return functions.

Given a function declared in a module, you can obtain a reference to it using the ^ operator. Once a reference has been assigned to a function value, calls can be made to it, as in the following example:

```
module Foo

function hello = {
  println("Hello!")
}

function main = |args| {
  let f = ^hello
  f()       # prints "Hello!"
}
```

Of course, direct function declarations can be made, as in the following:

```
let f = |str| {
  println(">>> " + str)
}
f("Hello!")     # prints ">>> Hello!"
```

Also, functions that consist of a single expression can be expressed using the shorthand -> notation:

```
let f = |str| -> println(">>> " + str)
f("Hello!")     # prints ">>> Hello!"
```

Function references have methods that allow them to be manipulated, mainly to perform operations such as partial *application* and *composition*. The provided methods closely mimic those of the java.lang.invoke package APIs. Let's look at the following example:

```
let add = |a, b, c| -> a + b + c
let times = |a, b| -> a * b

let add_1_and_2 = add: insertArguments(0, 1, 2)
let twice = times: bindTo(2)
let f = add_1_and_2:
  andThen(twice):
  andThen(|n| -> println(n))

# prints "12"
f(3)
```

Now let's step through the code. add_1_and_2 is the result of doing a partial application of the arguments of add starting at position 0, which means that a and b get fixed to 1 and 2. The resulting function takes a single argument, which is then passed as the argument c of the original add function. bindTo also performs partial application, except that it applies to a single value on the first parameter. andThen composes functions; here f first adds 1 and 2 to the single argument, then passes the result to twice, and then passes it to the anonymous function that prints the result.

Another interesting feature of Golo functions is that they support named arguments:

```
function startServer(address, port, options) {
  # (...)
}

function run = {
  # (...)
  startServer(
    port=8080,
    options=list["--watch", "--verbose"],
    address="0.0.0.0")
}
```

Named arguments are useful for clarifying the purpose of the parameters while using certain APIs. They also allow the order of the parameters to be different from that of the function declaration. All Golo functions, including anonymous functions, allow invocations with named arguments.

It is also worth noting that Golo can use named arguments on Java class methods. There is one limitation, though: classes need to have been compiled with the javac -parameters flag activated, which is, sadly, not the case by default.

Golo Functions and Java Functional Interfaces

Although lambdas were added only in Java 8, the language and platform have long relied on single-method interfaces. Typical examples include passing a Runnable instance to a Thread constructor or passing an ActionListener to a Swing component.

To explore a concrete example, let's use the java.util.concurrent.CompletableFuture class that was added in Java 8:

```
module SmiAndLambda

import java.util.concurrent.CompletableFuture

function main = |args| {
  supplyAsync(-> 1):
    thenApply(|n| -> n + 99):
    thenAcceptAsync(|n| -> println(">>> " + n)):
    join()
}
```

The code above creates a CompletableFuture by asynchronously executing a first function that does nothing interesting but returns 1. It then applies another function that increments the result of the first call by 99, and then asynchronously executes another function that prints out the result (>>> 1000).

The supplyAsync method from CompletableFuture accepts a java.util.function.Supplier, which is a single-method interface. The thenApply method expects a java.util.function.Function, which is a functional interface (that is, it has a single abstract method in conjunction with `default` methods). Similarly, `thenAcceptAsync` expects a `java.util.function.Consumer`, which is also a functional interface.

The Golo runtime automatically adapts function references to both Java single-method and functional interfaces.

Structures

Golo provides structured data definitions using the `struct` declaration, as in the following:

```
struct Message = { id, date, payload }
```

A structure object has a constructor, getter and setter accessors, as well as sensible `equals()`, `hashCode()`, and `toString()` methods. Here is a sample usage of the `Message` structure definition above:

```
let m = Message(
  id=12345,
  date="2016-01-22 15:41 CEST",
  payload="""I hope this message finds you well.

Yours sincerely,

- Julien""")

println(m)
m: id(6789)
println(m: id())
```

The code above would print:

```
struct Message{id=12345, date=2016-01-22 15:41 CEST,
payload=I hope this message finds you well.

Yours sincerely,

- Julien}
6789
```

Note that `"""` defines multiline strings in Golo.

Immutable copies of a `struct` object can be made by calling the `frozenCopy()` method. Every structure also comes with a constructor for making immutable objects: the name of the constructor function is prefixed by `Immutable`. In the previous example, we could have constructed `m` as immutable by calling `ImmutableMessage` rather than `Message`. This would also cause the `m: id(6789)` call to fail because `m` would be immutable.

Structure fields have public visibility by default. Given a module importing the module that defined the `Message` structure above, the module would have access to all fields through their accessors (for example, `id()` and `id(newValue)`). It is possible to restrict visibility by prefixing fields with an underscore. Any such field is visible only from within its defining module code, and remains hidden from other modules.

Finally, structure objects can be destructured and enumerated. This makes it possible to write the following:

```
let id, date, payload = m
println(id)
println(date)
println(payload)

foreach field, value in m {
  println(field + " -> " + value)
}
```

Tagged Unions

As a complement to structure types, Golo supports tagged unions, sometimes also called *sum algebraic data types*. Here is an example defining a `Figure` type:

```
union Figure = {
  Nothing
  Square = { sideLength }
  Rectangle = { firstSideLength, secondSideLength }
  Circle = { radius }
}
```

With this definition, a `Figure` can be of the `Nothing`, `Square`, `Rectangle`, or `Circle` concrete type. Each type can have (immutable) fields and a constructor. Here is the creation of a tuple with several `Figure` instances:

```
let figures = [
  Figure.Square(23),
  Figure.Rectangle(firstSideLength=10,
                   secondSideLength=20),
  Figure.Circle(30),
  Figure.Nothing()
]
```

Given each concrete type of `Figure`, methods are being provided to check whether an instance is of a given type: `f: isSquare()`, `f: isCircle()`, and so on. The following example illustrates how to use these methods. It also introduces the `match` operator that iteratively evaluates several conditions with `when`/`then`/`otherwise` clauses and returns a value:

```
println(figures: map(|f| -> match {
  when f: isRectangle()
    then
      "[ " +
      f: firstSideLength() +
      ", " +
      f: secondSideLength() +
      " ]"
  when f: isSquare()
    then "[ " +
      f: sideLength() + " ]"
  when f: isCircle()
    then "( " +
      f: radius() + " )"
  otherwise "."
}): join("\n"))
```

Golo collections provide functional idioms such as `map` (to create a new collection by applying a function to its elements) and `join` (to produce a string by concatenating elements with a separator). Running the code above prints the following text:

```
[ 23 ]
[ 10, 20 ]
( 30 )
.
```

It is also possible to test instances not just for their type, but also for values:

```
# false
println(Figure.Square(23): isSquare(20))

# true
println(Figure.Square(23): isSquare(23))
```

Because we might be interested in only a subset of a union of type fields, we can use the special `Unknown.get()` value to indicate that the value of certain fields is not useful for matching:

```
let figs = [
  Figure.Rectangle(10, 10),
  Figure.Rectangle(10, 30),
  Figure.Rectangle(30, 30),
  Figure.Circle(20)
]
let _ = Unknown.get()
```

```
println(figs: filter(|f| -> match {
  when f: isRectangle(10, _)
    then true
  otherwise
    false
}): join("\n"))
```

Starting with the `figs` collection, we can use `filter` to discard the figures that are not rectangles whose first side has a value of 10, and then print the result:

```
union Figure.Rectangle{firstSideLength=10,
    secondSideLength=10}
union Figure.Rectangle{firstSideLength=10,
    secondSideLength=30}
```

Augmentations

Golo does not provide constructions for defining classes, but it provides a way to add methods to any type that it can manipulate. This includes class definitions from Java APIs and also Golo `struct` and `union` types. An *augmentation* defines a set of functions that can be called as methods. The convention for these functions is to call `this` as the first parameter because it references the receiver object, but you are free to use a different name.

Augmentations can be defined by specifying a type:

```
augment java.lang.String {

  function wrap = |this, s1, s2| ->
    s1 + this + s2

  # (... more methods can be added)
}
```

The code above adds a `wrap` method to Java `String` instances. For example, the following code would give `"{abc}"`.

```
"abc": wrap("{", "}")
```

An augmentation applies to a type and all its subtypes; an augmentation on `java.lang.Object` would apply to every type. An augmentation is visible from its defining module and the modules that import this module.

The other way to define an augmentation is by name rather than by a target type:

```
augmentation Wrap = {
  function wrap = |this, s1, s2| ->
    s1 + this: pretty() + s2
}

augmentation PrettyContact = {
  function pretty = |this| ->
    this: name() + " <" +
    this: email() + ">"
}
```

With these two named augmentation definitions, we can compose and then augment the following `struct` type:

```
struct Contact = { name, email }

augment Contact with Wrap, PrettyContact
```

We can then use the augmented type as follows:

```
let dan = Contact("Dan", "dan@tld")
println(dan: pretty())
println(dan: wrap("/* ", " */"))
```

The code above prints:

```
Dan <dan@tld>
/* Dan <dan@tld> */
```

The advantage of named augmentations over augmentations on types is that you avoid code duplication when augmenting several, possibly unrelated, types.

Generating Adapter Classes

While Golo has few issues when calling Java APIs, there are cases when these APIs expect arguments to be objects that extend certain interfaces or base classes. Golo provides an API to generate *adapter classes*. An adapter class extends a base class, implements a set of interfaces, and overrides and implements methods.

The Adapter API is available through the gololang.Adapters import statement, and then it can be used to create objects such as the following one:

```
let obj = Adapter():
  extends("java.lang.Object"):
  interfaces([
    "java.io.Serializable",
    "java.util.Enumeration"]):
  implements("hasMoreElements",
    |this| -> true):
  implements("nextElement",
    |this| -> 100):
  overrides("toString",
    |this, super| -> "Strange!"):
  newInstance()

println(obj: getClass())
println(obj: getClass():
             getInterfaces():
             toString())
println(obj: hasMoreElements())
println(obj: nextElement())
println(obj: toString())
```

Executing the code above prints:

```
class $Golo$Adapter$0
[interface gololang.GoloAdapter,
  interface java.io.Serializable,
  interface java.util.Enumeration]
true
100
Strange!
```

The API performs soundness verifications (for example, it checks that all interface methods are being implemented) and dynamically generates the adapter class as JVM bytecode. Instead of method names, it is also possible to use * so that a single function can implement many methods or override all methods. The Golo documentation has examples of using adapters for dynamically generating proxies.

Calling Golo from Java

Calling Golo from Java is usually very easy. There are two options: direct invocations and code evaluations.

Direct invocations. The first option is to compile Golo source files, and add the generated bytecode plus the Golo runtime JAR file and dependencies to the Java application classpath. From the point of view of a Java class, a Golo module is a class with static methods. Given the following module:

```
module my.great.Module

struct Point = {x, y}

function a = |n| -> n + 1
function b = |a, b| -> a * b
```

The javap decompiler reveals the following public methods:

```
$ javap my.great.Module
Compiled from "compil.golo"
public class my.great.Module {
  public static java.lang.String[] $imports();
  public static java.lang.String[] $augmentations();
  public static java.lang.String[]
      $augmentationApplications();
  public static java.lang.String[]
      $augmentationApplications(int);
  public static java.lang.Object
      a(java.lang.Object);
  public static java.lang.Object
      b(java.lang.Object, java.lang.Object);
  public static java.lang.Object Point();
  public static java.lang.Object
      Point(java.lang.Object, java.lang.Object);
  public static java.lang.Object
      ImmutablePoint(java.lang.Object,
                     java.lang.Object);
}
```

[Indented lines are wrapped from the previous line. —*Ed.*] Methods such as `a` or `Point` can be called "as is" from Java. The methods prefixed with `$` are used by the Golo runtime.

Code evaluation. Another option is to use the `gololang.EvaluationEnvironment` class that is part of the Golo runtime. It provides various means to pass Golo code as strings, and either evaluate or execute them.

Here is an example usage of that class from Java:

```
package sample;

import gololang.EvaluationEnvironment;
import gololang.FunctionReference;

public class GoloFromJava {

  public static void main(String[] args)
    throws Throwable {
      EvaluationEnvironment env =
        new EvaluationEnvironment();

      FunctionReference f1 =
        (FunctionReference) env.def(
          "|a, b| -> println(a + \" ~ \" + b)");
      f1.invoke("hello", "world");

      FunctionReference f2 = (FunctionReference)
        env.asFunction("println(a + b)", "a", "b");
      f2.invoke(5, 10);
  }
}
```

The code above prints the following:

```
hello ~ world
15
```

The `EvaluationEnvironment` class provides additional methods for evaluating Golo code, such as evaluating the whole module source code from text, defining imports, and more.

Conclusion

This article introduced some interesting features of the Golo programming language. There is, of course, more to the language, and I encourage you to go further by experimenting with it. If you have always wondered how programming languages could be implemented, you might also enjoy looking at the source code: I generally do my best to ensure that it provides significant pedagogical value.

Last but not least, Golo is a language that is friendly to hobbyists. Do not hesitate to propose contributions! </article>

Julien Ponge (@jponge) is a longtime open source contributor who is currently an associate professor of computer science and engineering at INSA de Lyon, France. He focuses his research on programming languages, virtual machines, and middleware.

The author would like to thank the contributors of Golo, and especially Yannick Loiseau, Sylvain Desgrais, Daniel Petisme, and Philippe Charrière, who influenced the design and implementation of some language features.

SIMON ROBERTS

Quiz Yourself

More questions from an author of the Java certification tests

I hope you like the new format of these quizzes with longer and deeper explanations of the answers. Here are some more questions that simulate those from the 1Z0-809 Programmer II exam.

Question 1. Given this code:

```
public enum Suit {                  // line n1
  HEART, DIAMOND, SPADE, CLUB;      // line n2
  private Color color;              // line n3
  public final Suit(Color color) { // line n4
    this.color = color;
  }
}
```

Which two changes are necessary to enable the code to compile? Choose two.

a. Remove the keyword `public` from line n4.
b. Remove both the keywords `public` and `final` from line n4.
c. Remove the keyword `public` from line n1.
d. Change line n2 to this:

```
HEART(Color.RED), DIAMOND(Color.RED),
    SPADE(Color.BLACK), CLUB(Color.BLACK);
```

e. Change line n2 to this:

```
new HEART(Color.RED), new DIAMOND(Color.RED),
    new SPADE(Color.BLACK), new CLUB(Color.BLACK);
```

Question 2. You are creating a method that performs I/O operations that might throw an `IOException`. The I/O code is omitted from the fragments below, but it is assumed to occur at the location marked `// ...`

The `FileReader` constructor is declared as `throws FileNotFoundException`. The `FileNotFoundException` is a subclass of `IOException`. The code should respond to either exception by logging the exception and then rethrowing it to the caller.

Which code satisfies the requirements?

a.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
  // ...
} catch (IOException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
} catch (FileNotFoundException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
}
```

b.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
  // ...
} catch (FileNotFoundException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
} catch (IOException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
} finally {
  br.close();
}
```

c.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
```

```
  // ...
} catch (IOException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
}
```

d.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
  // ...
} catch (IOException | FileNotFoundException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
}
```

e.

```
try (BufferedReader br =
  new BufferedReader(new FileReader(fName));) {
  // ...
} catch (IOException | FileNotFoundException e) {
  LOG.warning(()->e.getLocalizedMessage());
  throw e;
} finally {
  br.close();
}
```

Question 3. Assume that the class `Fruit` is accessible and defines a JavaBeans-style accessor method with the following prototype:

```
public String getColor()
```

Given that `sf` is a nonempty `Stream<Fruit>` and given this code fragment:

```
System.out.println(
  // line n1
  .filter("yellow"::equalsIgnoreCase)
  .count());
```

Which of the following, when inserted at line n1, causes the fragment to output the total number of yellow fruits in the stream?

a.
```
sf.filter(f->f.getColor()
              .equalsIgnoreCase("yellow"))
```
b. `sf.flatMap(f->f.getColor())`
c. `sf.reduce(f->f.getColor())`
d. `sf.map(Fruit::getColor)`
e. None of these

Question 1. The correct answers are Options B and D. A significant part of the purpose of an enumerated type is to ensure that a specific set of instances, defined at compilation time, exists while the program is running. The only reason to provide access to a constructor is to allow the creation of new instances. Because new instances of an enum would break the expectations of the enum, there's no reason to allow access to the constructors. Consequently, as one of several measures to prevent such inconsistency, enum constructors must be `private`. A default constructor for an enum always will be `private`, and any constructor that lacks an explicit accessibility keyword will also be `private`. If an access control keyword is provided, it must be `private`.

Section 8.9 of the *Java Language Specification* states, "An enum type has no instances other than those defined by its enum constants. It is a compile-time error to attempt to explicitly instantiate an enum type." Section 8.9.2 notes, "In an enum declaration, a constructor declaration with no access modifiers is `private`." Finally, section 8.9.3 says, "It is a compile-time error if a constructor declaration in an enum declaration is `public` or `protected`."

Given this, it's clear that the public modifier must be removed from the constructor. However, while this is necessary, it is not sufficient. Let's see why. In general, constructors may not be final (they are not inherited, so it makes no sense to use this modifier).

Section 8.8.3 states, "Unlike methods, a constructor cannot be abstract, static, final, native, strictfp, or synchronized."

So, this tells you that it's necessary to remove *both* the public and final keywords from the constructor. The only option that provides for this requirement is Option B, so this must be part of the correct answer. This solution also eliminates Option A.

Option C calls for removing the public keyword from the enum class as a whole. This is unnecessary; it's perfectly correct to have a public enum.

Section 8.1.1 mentions that the class modifiers of a normal class may include public. Later, section 8.9 modifies the information to be specific to enums. It prohibits abstract and final, but it does not prohibit the others, ensuring that public is acceptable. Because of this, Option C must be incorrect.

Lastly, when an explicit constructor is defined for any class, the default constructor is not generated by the compiler (section 8.8.9). Because of this, line n2, as shown in the original, will not compile. This is because the format it uses has no parentheses and no argument lists; therefore, it attempts, unsuccessfully, to invoke that missing default constructor. Section 8.9.1 notes that when an argument list is provided, these arguments are "passed to the constructor of the enum." It also notes that normal overload-matching rules will be applied. Because of this and the format specified in this same section, the proper format for the invocation is as shown in Option E. The form in Option E that uses a new keyword is a syntax error. Because of this, Option D is the second correct option, and Option E is wrong.

Note for certification exam students: You might wonder how an item can be correct when it refers to Color.RED and Color.BLACK in the argument to the constructor, but it does not define or import any Color class.

In the interest of keeping the amount of code you have to examine within reasonable limits, Oracle has documented some assumptions that should be made when considering a question. These are listed on the exam information pages. In particular, the following assumptions are mentioned explicitly. "Missing package and import statements: If sample code does not include package or import statements, and the question does not explicitly refer to these missing statements, then assume that all sample code is in the same package, and import statements exist to support them." Given this, it's clear that it's proper to assume that some class Color exists, and it provides for these constants. If you're unfamiliar with it, this is probably the original java.awt.Color class, but such detail isn't important here.

Most of the notes that Oracle has provided might be considered obvious. That is, if you didn't make these assumptions, many questions would have no plausible answer. However, it's nice that they're now called out explicitly, so you don't have to worry if you're making an unreasonable assumption. You can find these notes on the pages of each relevant exam, usually by selecting the **Exam Topics** tab that's about halfway down.

Question 2. The correct answer is Option C. FileNotFoundException is a subclass of IOException. Where exceptions in a class hierarchy are both being caught explicitly—such as in Option A—the more-specific exception must be positioned earlier in the list than the more-general exception. This ordering rule exists because execution will jump to the *first* catch block that is applicable (section 14.20.1 of the *Java Language Specification*), and if the more-general were first, the more-specific would never be executed. In Option A, this rule is broken; hence, the code fails to compile and is, therefore, incorrect.

It's perhaps interesting to note that in earlier versions of Java, that's exactly what happened, leading to some potentially hard-to-find bugs. Today, section 14.21 identifies such a situation as *unreachable code* and requires the compilation error that we're used to now.

In Option B, the ordering of the `catch` blocks has been corrected; however, an explicit `finally` block has also been added. It's OK to have a `finally` block, but the scope of the resource variable `br` (the `BufferedReader`) is limited to the parentheses following the `try` statement and the block that follows it. Consequently, the attempt to explicitly close `br` in the `finally` block fails to compile. Of course, the whole point of the `try`-with-resources structure is to close the resources implicitly, so attempting to close `br` explicitly like this is misguided, too.

Option C works just fine, and turns out to be the correct answer. Because the `FileNotFoundException` is a subclass of `IOException`, they'll both be caught in the single `catch` block provided, and they will be treated as the specification demands. It's perhaps worth discussing that if I wanted different handling for these two exceptions, or if I wanted the same handling for two exceptions that did not share a parent/child relationship, this wouldn't be a good solution. But that's not the situation in this question.

Option D might look tempting, using the newer (Java 7) multi-`catch` syntax. However, it fails because the two exceptions share a parent/child relationship. This code actually creates a compilation error. Section 14.20 notes, "It is a compile-time error if a union of types contains two alternatives `Di` and `Dj (i ≠ j)` where `Di` is a subtype of `Dj`." That's a bit of a mouthful, but in essence it simply excludes using the multi-`catch` syntax with exception types, such as `IOException` and `FileNotFoundException`, that have an inheritance relationship.

Option E fails for the same reasons as both Options D and B.

A couple of side notes: First, I think a good case could be made that the exam might not explicitly tell you about the inheritance relationship between these two exceptions, given how common they are, and how many other objectives hint at knowing this kind of detail.

Second, you'll notice that there's a lot of code in this example. While questions of this size are not common, you will sometimes come across them. It's not a bad skill to practice keeping a clear head and making an organized search to look for particular points. After all, production code isn't always written and maintained in the best conventions of clean code either!

Question 3. The correct answer is Option D. For this fragment to output the number of occurrences of yellow fruits, the stream that feeds into the `count` method (which is itself the output of the `filter` method) must contain one element for each yellow fruit in the original stream. It doesn't particularly matter what that element is, of course.

The `filter` method that is already in place will pass through only those elements which are strings that match (ignoring case) the text `"yellow"`. This means that the stream that enters the `filter` method must contain only the color names of the original `Fruit` objects, not the `Fruit` objects themselves.

The method in the `Stream` interface that allows you to convert the contents of a stream—either the values or the data type—while maintaining a one-to-one correspondence between input and output items is `map`. Option D does this, and is correctly formed, using a method reference to the `getColor` method to take a `Fruit` object from the input stream and extract the color name. That, as was just discussed, is what's needed for this code to behave as required. So, Option D is correct.

Option A would work if it *replaced* the existing `filter` statement. This statement would result in the yellow `Fruit` objects proceeding down the stream. If counted directly, we'd get the desired result, but, as the code stands, the downstream filter would fail to compile because it's receiving whole `Fruit` objects, rather than color names.

Option B would not compile, because the return type of the behavior provided as the argument to `flatMap` must be a stream of some kind. It's interesting to note, however, that if the option had been `sf.flatMap(f->Stream.of(f.getColor()))`, then—although horribly inefficient and cumbersome—it would have worked correctly.

Option C also fails to compile and is essentially nonsense. The `reduce` method of `Stream` is a terminal operation that collects all the values coming down the stream and produces a single output value. The behavior argument to the `reduce` method performs the computations that create that single output value. Clearly, that's not what's needed here anyway. Further, the behavior argument to the single-argument `reduce` method is a `BinaryOperator`, and the lambda `f-> f.getColor()` is effectively a `Function<Fruit, String>` and is, therefore, incompatible. `</article>`

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies in Silicon Valley and around the world. He remains involved with Oracle's Java certification projects.

learn more

[Understanding enums](#)
[Oracle Java tutorial on FileReader](#)
[Oracle Java tutorial on Streams and aggregate operations](#)

FEATURED JDK ENHANCEMENT PROPOSAL

JEP 283 and JEP 263: Migrating to GTK+ 3 on Linux

Because this issue focuses on what's inside Java, let's look at how Java renders user interface elements on Linux. At the moment, the Linux version of the JRE depends in part on the GIMP Toolkit (GTK+), a portable UI library written in C. However, Java depends on GTK+ 2, a rather old version of the library. JEP 283 proposes migration to GTK+ 3. The benefit is that GTK+ 3 is the development branch (and GTK+ 2 is no longer active). Another driver for this change is a problem that could potentially arise, namely lack of future support for GTK+ 2 in Linux distros. Currently, most versions of Linux ship with both GTK+ 2 and 3. However, because version 3 was launched in 2011, it's not clear how long the various distros will keep bundling GTK+ 2. This enhancement proposal, therefore, explores how support for version 3 could begin to be brought into AWT/Swing, JavaFX, and potentially SWT. The JDK Enhancement Proposal (JEP) document is particularly interesting, because it presents a good overview of how much work has to be done to make a comparatively small change to a single component on a single platform. It explains the benefits and drawbacks well, and it lays out various transitional paths.

An earlier proposal, JEP 263, pointed to the benefits of GTK+ 3 as a library to use for better support of high-resolution displays.

If you're interested in this topic and have expertise in the matter, the Expert Committees welcome your input.

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.

☛ Download area for code and other items

☛ *Java Magazine* in Japanese